(2)

Document No. 2020-001
30 June 1990

User's Manual for A Prototype Binding of ANSI-Standard SQL to Ada Supporting the SAME Methodology for the Software Technology for Adaptable, Reliable Systems (STARS) Program

Contract No. F19628-88-D-0032

Task IR67 - Prototype Binding of ANSI-Standard SQL

CDRL Sequence No. 2020

30 June 1990

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

IBM Federal Sector Division
800 North Frederick Avenue
Gaithersburg, MD 20879

90

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 30, 1990	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE User's Manual for a Prototype Binding of ANSI-Standard SQL to Ada Supporting the SAME Methodology	5. FUNDING NUMBERS C: F19628-88-D-0032
6. AUTHOR(S) S. Phillips	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IBM Federal Sector Division 800 N. Frederick Avenue Gaithersburg, MD 20879	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Electronic Systems Division Air Force Systems Command, USAF Hanscom AFB, MA 01731-5000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER CDRL Sequence No. 2020

11. SUPPLEMENTARY NOTES

12a. *DISTRIBUTION/AVAILABILITY STATEMENT*	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A standard binding was needed between Ada and SQL (Structured Query Language), the ANSI and DoD standard for accessing commercial relational data base management systems (DBMS's). SQL was not designed to be embedded within applications in general purpose programming languages, such as Ada. Previously developed Ada-SQL bindings have had various technical drawbacks.

A prototype Ada-SQL binding was built by automating the SQL Ada Module Extension methodology (SAME). SAME is a method for building Ada applications that access DBMS's via SQL. SAME extends SQL by exploiting the features of Ada.

This User's Manual contains ~~installation~~ instructions, compilation order, guidelines for input data, detailed steps to create a specific binding, and information about porting the system to another DBMS.

[illegible] STARS (Software [illegible]) [illegible]

14. SUBJECT TERMS STARS, Ada, SQL, Structured Query Language, data base management system, DBMS, SAME	15. NUMBER OF PAGES 145
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

LMSC-F376702

USER'S MANUAL

for

A Prototype Binding of ANSI-Standard SQL to Ada Supporting the SAME Methodology

Ada SQL

Contract Number: F19628-88-D-0032/0002

CDRL Number: C2020

Submitted to

IBM Corporation
Systems Integration Division
800 N. Frederick Road
Gaithersburg, Maryland 20879

June 7, 1990

Lockheed
Missiles & Space Company, Inc.
Software Technology Center
2100 East St. Elmo Road
Org. 96-10 / Bldg. 30E
Austin, TX 78744
(512) 448-5740

E-mail: phillips@stc.lockheed.com

RELEASE PAGE

This manual is compatible with Version 1 of **A Prototype Binding of ANSI- Standard SQL to Ada Supporting the SAME Methodology**, released June 7, 1990.

RELEASE PAGE

This manual is compatible with Version 1 of **A Prototype Binding of ANSI- Standard SQL to Ada Supporting the SAME Methodology**, released June 7, 1990.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Section 1
Overview

1.1 INTRODUCTION

This manual is intended for users of the Ada/SQL binding supporting the SQL Ada Module Extensions (SAME) methodology developed by Lockheed Software Technology Center. Ada is a programming language resulting from a collaborative effort to design a common language for developing large-scale, real-time systems. Structured Query Language (SQL) consists of a set of facilities for defining, manipulating, and controlling data in a relational data base. The SAME approach as described in *Guidelines for the Use of the SAME*, an SEI Technical Report [1], is a method for the construction of Ada applications that access data base management systems (DBMS) whose data manipulation language is SQL.

Conventional approaches for binding ANSI-standard SQL to Ada allow the embedding of SQL statements directly into Ada programs, thereby creating something that is neither pure SQL nor pure Ada. A preprocessor is used to remove the SQL statements and replace them with valid Ada subprogram calls. However, direct access to the data base is still part of the application program. Using Lockheed's SAME approach, the SQL statements and Ada statements are separate, providing a modular approach to data base definition and access and allowing the efficiency of having programming tasks assigned to programmers who specialize in each area.

As its name implies, SAME extends the module language defined in the ANSI SQL standard *Database Language—SQL* [2] by exploiting the capabilities of Ada. The defining characteristic of the module language is the collocation of SQL statements, physically separated from the Ada application, in an object called the *concrete module*. SAME allows an Ada application to treat the module much the same as it treats any other foreign language; that is, it imports complete modules, not language fragments. SAME provides the binding between these two modules through an interface layer—called the *abstract interface*—that consists of an Ada specification and body. The abstract interface transforms data from abstract definitions to concrete types (and back again). The abstract interface makes calls to an Ada specification representative of the SQL module. Both the SQL module and its representative Ada specification are called the *concrete interface*. All abstract data types supporting this interface are contained in abstract domain packages.

The Ada/SQL binding tool provides automatic generation of the abstract interface, the specification part of the concrete interface and the abstract domain packages. This binding also allows Ada compile-time checking of all data manipulation procedures within a user's application program. The approach uses no preprocessor, and all code is valid Ada, compilable under any validated Ada compiler. The binding may be associated with any commercial off-the-shelf data base. For development, the DEC RDB data base management system executing on a Vax workstation running the VMS operating system was used. At present, this is the only commercially available platform that has an SQL module language compiler. A module language compiler can be simulated using one of many DBMS systems that support embedded SQL. When associating the binding with another data base, some implementation details will change. Included in this manual is a description of the support provided by this implementation to facilitate those changes.

1.2 THE PURPOSE OF THIS MANUAL

The purpose of this manual is to provide the guidance needed to use tools to define and generate a SAME binding for a specific data base application. This manual offers assistance for all types of users during development of an interface and the accompanying application programs. It explains the methodologies and strategies used to define and generate an interface and to use the interface with an Ada application. This explanation is given as a series of well-defined steps. These steps are outlined in Sections 3 through 6. This manual is intended for use by the following people:

- The interface programmer—to define the abstract domain packages, the abstract interface, and the concrete interface.
- The application programmer—to use the binding and have compile-time checking of all data manipulation statements within the Ada program.
- The system operator—to install the system on any machine using any data base and modify it for use with different data base implementations.

This manual is not intended to be either an SQL or SAME reference manual, and the audience of this manual is assumed to be familiar with both SQL and the SAME method. Refer to either *Guidelines for the Use of the SAME* or the ANSI SQL standard *Database Language—SQL* for more information about SQL and the SAME method.

1.3 THE SCOPE OF THIS MANUAL

This manual is intended as a guide to installing and using an implementation of an Ada/SQL binding. The manual does not define specific uses for this binding nor any of the reasons that a binding is necessary. The manual does not provide design decisions made during development or give any rationale to the choices made during implementation. Finally, this manual is written for persons knowledgeable of ANSI-standard Ada, the SQL data manipulation and definition language, and the SAME method. Explanations of what specific SQL statements do (i.e., What is an update statement?) or detailed explanations of the SAME method (i.e., How are abstract domains determined?) are not provided. The manual can be used by all members of a team who are responsible for writing an Ada application program that accesses a data base with an SQL interface. The manual provides instruction at each step in this process.

1.4 MANUAL ORGANIZATION AND CONTENT

This manual is divided into five main sections and five appendices. Section 1 provides an overview of the binding and manual content. Section 2 briefly discusses the SAME approach and the major functionality of the delivered software. Included in this section are graphic representations of both the SAME approach and interface creation and architecture. Sections 3 and 4 are intended for use by the interface programmer. Section 3 provides instructions for defining and generating the abstract domains, while Section 4 explains the methodology for defining an interface and generating the abstract interface and the Ada specification part of the concrete module. Section 5 is used by the Ada application programmer. It provides the user with the names of all packages that must be imported by the application to use the binding. A description of these packages is also included. Section 6 is intended for use by system personnel who are responsible for installing the software for the binding. Included in this section are instructions for associating the binding with another data base.

Appendix A provides a more detailed description of the program's components with each major component within the system discussed in terms of inputs, outputs, and functionality. This appendix also contains the compilation order of all components. Appendix B lists all program exclusions of the SAME method. Errors, error messages, and suggestions for correction are discussed in Appendix C. Comprehensive sample programs illustrating domain and interface definition, generation, and use are provided in Appendix D. Appendix E contains a reference list.

1.5 MANUAL TYPESET AND NOTATION CONVENTIONS

Throughout this manual, the following typeset and notation conventions are used to increase clarity:

- `Courier Typeface` denotes actual filenames or Ada unit names and Ada code.

  Examples:
  `[project]test_instructions.txt`—Full name of a text file
  `Abstract_Interface_Generator`—An actual Ada unit
  `type records is (first_record, second_record);`—Ada code

- `COURIER CAPITALS` denote SQL module language commands and code.

  Examples:
  `CLOSE CURSOR`—SQL module language command
  `SELECT AGE FROM EMPLOYEE`—SQL module language code

- `{Courier typeset enclosed in brackets}` denotes VMS or RDB commands and code. Note that the brackets are not part of the code.

  Examples:
  `{set def [directory_name]}` —A VMS command
  `{create table}` —An RDB command

- <*Key* > denotes a key.

  Example:
  <*Return* > —The return key

- *Italics* denote user-supplied names, expressions, and commands.

  Example:
  Procedure_Name —A specific procedure to be named by the user (e.g., substitute "Update_Row" for *Procedure_Name*)

Section 2
Computer Program System Capabilities

2.1 PROGRAM PURPOSE

The purpose of this ADA/SQL binding is to provide tools and methodologies to assist in the construction of a SAME Ada/SQL interface to a commercial SQL data base. This program is to be used for the automatic generation of the SAME-required domain packages, abstract interface, and Ada specification part of the concrete interface from two procedures, a domain view procedure and an interface view procedure. These products can then be `withed` by an Ada application programmer. This approach requires no preprocessing of source code in any language and is compatible with any commercial Ada compiler. Note that this program does not assist in actual data base creation and assumes that a data base exists and its structure and content are known.

2.2 GENERAL DESCRIPTION OF THE SAME APPROACH

During application design and development, the SAME method is used as shown here. The automatic support provided by this binding tool is also indicated.

- The application programmer, along with the SQL programmer, determines the services that will be needed from the data base. They are coded in SQL and collected in an SQL module called the concrete module.
- The abstract domains that occupy data base columns are defined and described as Ada types. This is done using standard packages available to users of SAME methodology. The domain packages are specific to a given data base but not necessarily to a given application (more than one application can be supported by the packages). Automatic support is provided for domain generation.
- An abstract interface is created. This is a set of package specifications declaring the record type definitions needed to describe row records and the procedure declarations needed to access the relevant concrete module procedures. This interface will be called by the Ada application. Unlike the domain packages, the interface is specific to one application. Note that each application can use multiple abstract interfaces. Automatic support is provided for abstract interface generation.
- The application program is written. Once the abstract interface specification is completed, the application program can be written concurrently with the creation of the abstract module bodies because the application does not need the module bodies to compile.
- The abstract module—the bodies of the procedures declared in the abstract interface—is created. Automatic support is provided for abstract module generation.

Figure 2-1 shows the configuration of an Ada application accessing a DBMS using SAME. Dashed, two-way arrows indicate data flow and solid, one-way arrows indicate Ada visibility.

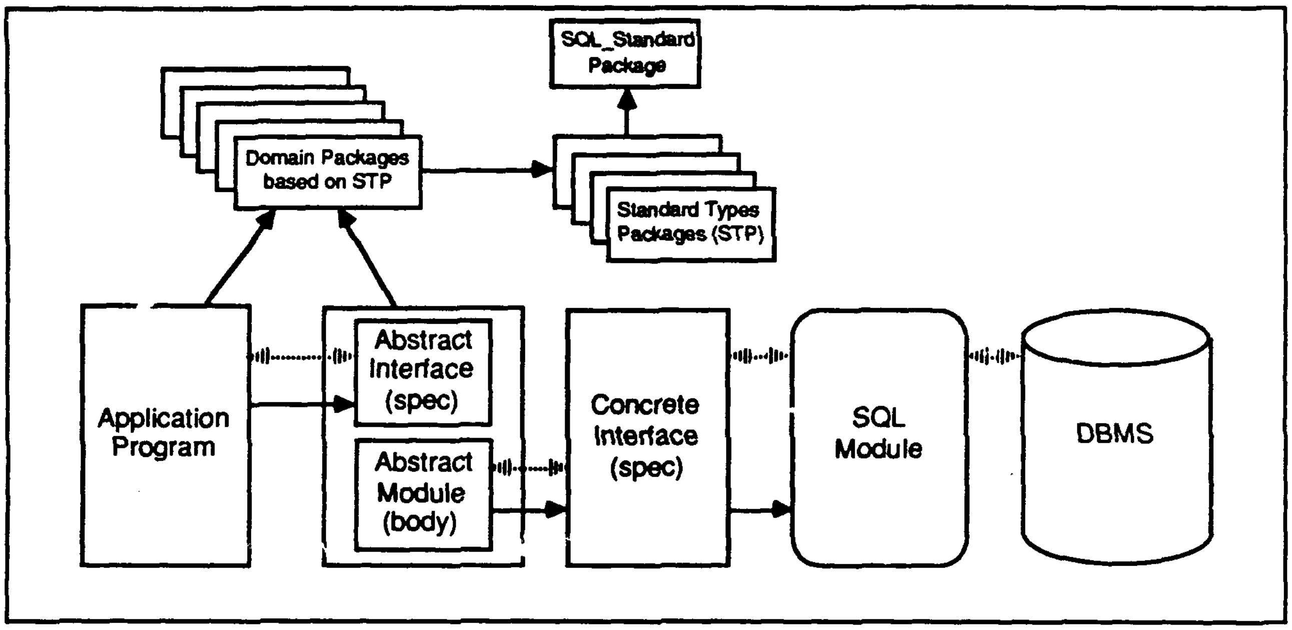
SQL_Standard Package
Domain Packages based on STP
Standard Types Packages (STP)
Application Program
Abstract Interface (spec)
Abstract Module (body)
Concrete Interface (spec)
SQL Module
DBMS

Figure 2-1. Ada Application Accessing a DBMS Using SAME

2.3 GENERAL DESCRIPTION OF PROGRAM

This program can be conceptually thought of as two tools, one for SAME domain package generation and one for SAME abstract interface and concrete interface generation. Figure 2-2 provides a graphic overview of the binding's architecture and the ordered steps (steps 1-4) for creating the interface. Hashed arrows represent automatic code generation, and solid arrows represent Ada visibility.

Step 1. The SQL programmer writes the SQL module containing all of the SQL data manipulation statements needed by the Ada application accessing the database. Once written, the SQL module is turned over to the interface programmer who will generate the corresponding domain packages and interface.

Step 2. The interface programmer writes an Ada procedure (Domain View in Figure 2-2) to define and generate the domain packages used by the application. Specific syntax and content for this procedure are given in Section 3. This procedure instantiates the generic package, `Abstract_Domain_Generator`, with several parameters that are used to define the name of the package containing the abstract domains, the names of the columns within the data base that are being defined as domain, and the attributes of each domain, such as length or type.

When this Ada procedure (Domain View) is compiled and executed, syntactically valid Ada packages that support SAME abstract domain semantics are generated automatically. These packages are compiled and made available to the application programmer. All generated domains are based on the data types contained in the SAME standard types packages and declared in the Ada specification `SQL_Standard`. This method provides enough redundancy to check for undefined or multiply defined domains (or columns). The interface programmer is notified of any errors in domain definition by means of error messages printed to the screen. The domain packages are generated only after a successful execution.

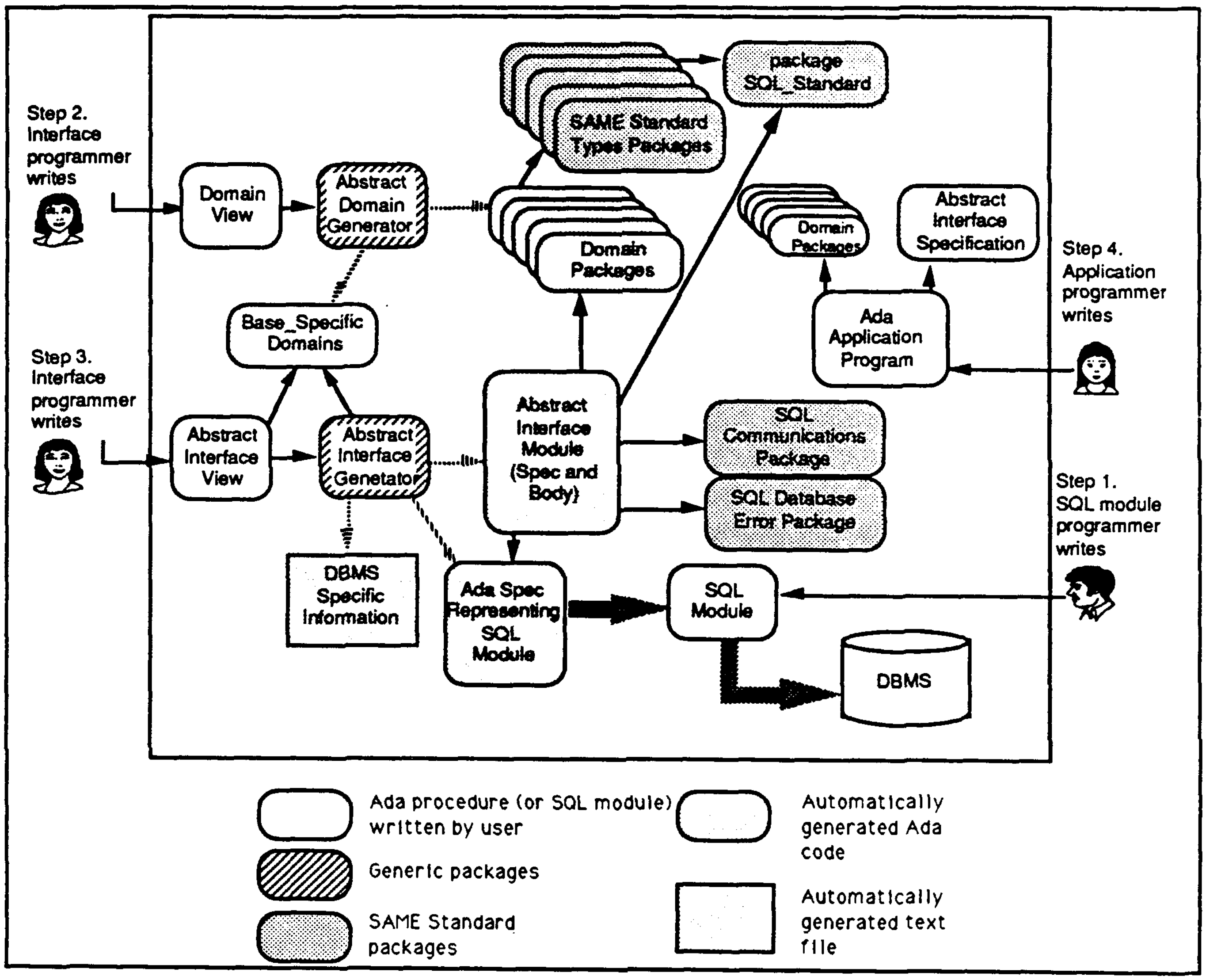
Step 2. Interface programmer writes
Domain View
Abstract Domain Generator
SAME Standard Types Packages
package SQL_Standard
Domain Packages
Domain Packages
Abstract Interface Specification
Step 4. Application programmer writes
Base_Specific Domains
Ada Application Program
Step 3. Interface programmer writes
Abstract Interface View
Abstract Interface Genetator
Abstract Interface Module (Spec and Body)
SQL Communications Package
SQL Database Error Package
Step 1. SQL module programmer writes
DBMS Specific Information
Ada Spec Representing SQL Module
SQL Module
DBMS
Ada procedure (or SQL module) written by user
Generic packages
SAME Standard packages
Automatically generated Ada code
Automatically generated text file

Figure 2-2. A Graphic Overview of the Binding's Architecture and the Ordered Steps (1-4) for Creating the Interface

In addition to the domain packages, an Ada package specification, `Base_Specific_Domains`, is generated automatically. This package is an Ada specification that contains an enumerated list of all valid domains and all valid domain packages to be used by the interface programmer when generating the abstract interface.

Step 3. The interface programmer writes another Ada procedure (Abstract Interface View in Figure 2-2), as detailed in Section 5, that generates the `Abstract_Interface`, both specification and body, and the Ada specification for the SQL module. This procedure, as well as `Abstract_Interface_Generator`, will `with` the previously generated Base_Specific_Domains package specification to ensure consistency between the abstract interface being generated and the domain packages previously generated. The interface programmer is notified of any errors in abstract interface definition by means of error messages printed to the screen. The abstract interface and Ada specification part of the concrete interface are generated only after a successful execution.

This procedure instantiates the generic package, `Abstract_Interface_Generator`, with parameters that define the name of the abstract interface package, the names and components of each row record, and the names and parameters of each procedure corresponding to the SQL procedures. When this Ada procedure is compiled and executed, syntactically valid Ada packages that support SAME abstract and concrete interface semantics are automatically generated. These packages are compiled and made available to the application programmer. Because the interface programmer's procedure provides the `Abstract_Interface_Generator` with enough semantic knowledge of the structure of the `Abstract_Interface`, most incomplete definitions, missing definitions, and semantically invalid definitions are caught at compilation.

In addition to the interface packages, a text file—`Dbms_Specific`—is generated automatically. This text file contains interface information that can be used to modify the SAME standard interface to support different DBMS implementation.

Step 4. The final phase deals with the application programmer, who has been given the Ada domain packages, the abstract interface and the Ada specification of the concrete interface, and the SQL concrete module. The programmer will import the interface and the domain packages into his program via `with` clauses. Once imported, the application program has access to the DBMS through the domains and procedures provided.

Section 3
Defining and Generating the Domains

3.1 GENERAL PREPARATION

For this prototype binding, it is assumed that all data base tables and views have been created. The data base structure for a hypothetical data base, `Dixie_Db`, shown in Table 3.1 will be the basis for later examples of interface development throughout Sections 3 through 5.

After data base creation, the application programmer informs the SQL programmer of the data base services that are required. Based on that information, the SQL programmer writes an SQL module containing all of the SQL statements needed by the Ada application. Guidelines and rules governing SQL module programming are defined in the ANSI SQL standard *Database Language —SQL* [2].

Table 3.2 shows an example SQL module named DIXIE_CONC_INTERFACE. This SQL module contains some example SQL statements for manipulating data in the DIXIE_DB data base.

Table 3.1. Structure for the DIXIE_DB Data Base

TABLE EMPLOYEES
EMPLOYEE_ID CHAR(5) NOT NULL
LAST_NAME CHAR(10) NOT NULL
FIRST_NAME CHAR(10) NOT NULL
ADDRESS CHAR(30) NOT NULL
WEIGHT REAL NOT NULL
TABLE JOB_HIST
EMPLOYEE_ID CHAR(5) NOT NULL
TITLE CHAR(15)
YEARS_EMPLYD INT
TABLE SALARY_HIST
EMPLOYEE_ID CHAR(5) NOT NULL
SALARY REAL NOT NULL
MERIT_PTS INT

Table 3.2. SQL Module `DIXIE_CONC_INTERFACE`

```
-- THIS SQL MODULE PROVIDES THE SQL STATEMENTS NEEDED BY THE
-- DIXIE_APPLICATION.ADA PROGRAM.
________________________________________________________________
-- HEADER INFORMATION SECTION
________________________________________________________________
MODULE          DIXIE_CONC_INTERFACE        -- MODULE NAME

LANGUAGE        ADA                         -- LANGUAGE OF CALLING PROGRAM

AUTHORIZATION   DIXIE_DB                    -- PROVIDES DEFAULT DB HANDLE
________________________________________________________________
-- DECLARE STATEMENTS SECTION
```

```
DECLARE SCHEMA FILENAME 'DIXIE_DB'        -- DECLARATION OF THE DATABASE

DECLARE EMPLROW CURSOR FOR
   SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, ADDRESS, WEIGHT FROM EMPLOYEES
   WHERE EMPLOYEE_ID = INPUT_EMPLOYEE_ID
```

```
-- PROCEDURE SECTION
```

```
-- THIS PROCEDURE USES THE DEC RDB EXECUTABLE FORM FOR STARTING A TRANSACTION
-- THIS PROCEDURE IS ONLY USED WITH THE DEC RDB DBMS
PROCEDURE SET_TRANSACTION
   SQLCODE;

   SET TRANSACTION READ WRITE;

-- THIS PROCEDURE OPENS THE CURSOR THAT HAS BEEN DECLARED FOR EMPLOYEES TABLE
PROCEDURE OPEN_EMPLOYEES_CURSOR
   INPUT_EMPLOYEE_ID CHAR(5)
   SQLCODE;

   OPEN EMPLROW;

-- THIS PROCEDURE FETCHES THE DATA FROM THE OPENED EMPLROW CURSOR
PROCEDURE FETCH_EMPLOYEES
   OUTPUT_EMPLOYEE_ID CHAR(5)
   OUTPUT_LAST_NAME CHAR(10)
   OUTPUT_FIRST_NAME CHAR(10)
   OUTPUT_ADDRESS CHAR(30)
   OUTPUT_WEIGHT REAL
   SQLCODE;

   FETCH EMPLROW INTO OUTPUT_EMPLOYEE_ID,
                      OUTPUT_LAST_NAME,
                      OUTPUT_FIRST_NAME,
                      OUTPUT_ADDRESS,
                      OUTPUT_WEIGHT;

-- THIS PROCEDURE UPDATES THE SALARY_HISTORY TABLE
   PROCEDURE INSERT_SAL_HIST_ROW
     INPUT_EMPLOYEE_ID CHAR(5)
     INPUT_SALARY REAL
     INPUT_MERIT_PTS INT MERIT_IND SMALLINT
     SQLCODE;
     INSERT INTO SALARY_HIST
     VALUES (INPUT_EMPLOYEE_ID, INPUT_SALARY, INPUT_MERIT_PTS MERIT_IND);

-- THIS PROCEDURE UPDATES THE JOB_HIST TABLE
PROCEDURE UPDATE_JOB_HIST
   INPUT_EMPLOYEE_ID CHAR(5)
   SQLCODE;

   UPDATE JOB_HIST
   SET YEARS_EMPLYD = YEARS_EMPLYD + 1
   WHERE EMPLOYEE_ID = INPUT_EMPLOYEE_ID;

-- THIS PROCEDURE COMMITS THE TRANSACTION
PROCEDURE COMMIT_TRANSACTION
   SQLCODE;

   COMMIT;

-- THIS PROCEDURE ROLLS BACK THE TRANSACTION
PROCEDURE ROLLBACK_TRANSACTION
   SQLCODE;

   ROLLBACK;

-- THIS PROCEDURE CLOSES THE EMPLROW CURSOR
PROCEDURE CLOSE_EMP_CURSOR
   SQLCODE;

   CLOSE  EMPLROW;
```

Once the SQL module is completed, the SQL programmer hands over the module to the interface programmer, and all further operational entry tasks in this section are the responsibility of the interface programmer.

Before beginning domain definition, the interface programmer creates a directory to contain the SQL module(s), the domain packages, the interface(s), and the Ada/SQL binding tool. A characteristic of the binding tool requires that only one set of domains packages is in a directory at any given time to avoid ambiguity. All components of the binding tool up to and including the `Abstract_Domain_Generator` should be compiled in the order shown in Appendix A.

3.2 INPUTS FOR DOMAIN DEFINITION AND GENERATION

Based on SQL statements coded in the SQL module, the interface programmer determines the abstract domains needed to support the interface. Guidelines for domain determination are explained in *Guidelines for the Use of the SAME* [1]. Next the interface programmer writes an

Ada procedure, *`Domain_View`*, to create the domain packages. This procedure instantiates several layers of nested generics contained in `Abstract_Domain_Generator`. When executed, it produces two or more text files, one or more Domain Packages, and the file `Base_Specific_Domains`.

The following paragraphs provide the steps and guidelines for writing the Ada procedure, `Domain_View`, that will generate these files.

3.3 STEPS AND GUIDELINES FOR PROCEDURE *Domain_View*

To define the domain packages, the interface programmer always follows the same basic steps. These steps take the form of writing an Ada procedure that instantiates each of the three levels of nested generics within the `Abstract_Domain_Generator` generic package. This does not mean that every procedure or generic package is used but that at least one generic package in each level of the nesting must be instantiated.

The following numbered steps are described in the order in which the interface programmer writes the Ada code. Each step includes example code that implements the step and a complete procedure template, also naming the steps, is shown in Table 3.3. Note that *italics* indicate user-supplied names or values. Names can be any valid Ada name but must be used consistently throughout the procedure.

Step 1. Import 2 packages:
`Abstract_Domain_Generator` => contains the generics that when instantiated generate the Domain packages and the file `Base_Specific_Domains`. This package must always be imported.

`Generator_Support` => contains common declarations (mostly enumerated types and constants). This package must always be imported.

```
with abstract_domain_generator;
with generator_support; use generator_support;
```

Step 2. Create an Ada main procedure.

```
procedure Domain_View is
  begin
end Domain_View;
```

Step 3. Within this procedure declare an enumerated type representing all valid names of domain packages to be generated. These names will also appear as the filenames of the generated packages.

Note: Duplicate package names are not allowed.

```
type Pack_Names  is
        (Package_Name1, Package_Name2, Package_Name3, ...);
```

Step 4. Instantiate the outermost generic package `Abstract_Domain_Generator` with the type representing the domain package names. From now on, only the package names represented by *Pack_Names* will be recognized as valid during elaboration of subsequent generic packages.

```
package Dom_Packs is new
        abstract_domain_generator(Pack_Names);
```

Step 5. Steps 5 through 8 are repeated once for each domain package, *Package_Name1*, *Package_Name2*, etc., enumerated in Step 3. For each domain package, declare an enumerated type representing all valid names of domains to be contained in that domain package. From now on, only these domain names will be recognized as valid during elaboration of subsequent generics.

Note: Duplicate domain names are not allowed.

```
declare
type Doms is
        (Domain_Name1, Domain_Name2, Domain_Name3, ...);
```

Step 6. Instantiate the second-level generic package, `Generate_Domain_Package`, with the specific domain package name and the enumerated type *Doms* that represents the domain names.

```
package Dom_1 is new
Dom_Packs.generate_domain_package(Package_Name1, Doms);
```

Step 7. Step 7 is repeated once for each domain, *Domain_Name1*, *Domain_ Name2*, etc., enumerated in Step 5. For each domain, instantiate a third-level generic package with parameters that describe the domain. Which generic and what parameters depend on the type of domain. An example of each one of the six possible generic instantiations is shown below, however, more detailed descriptions of each generic along with parameter explanations are found in the next section.

```
--  For domains of type char, int or small int.
package First is new Dom_1.generate_int_domain
        (Domain_Name1, Domain_Type, Null_Indicator,
             Range_Start, Range_Stop);

--  For subtypes of domains of type char, int or small int.
package Second is new Dom_1.generate_subint_domain
        (Domain_Name2, Domain_Based_On, Domain_Type,
             Null_Indicator, Range_Start, Range_Stop);

--  For domains of type real, decimal or double_precision.
package Third is new Dom_1.generate_flt_domain
        (Domain_Name3, Domain_Type, Null_Indicator,
             Range_Start, Range_Stop);

--  For subtypes of domains of type real, decimal or
--  double_precision.
package Fourth is new Dom_1.generate_subflt_domain
        (Domain_Name4, Domain_Based_On, Domain_Type,
             Null_Indicator, Range_Start, Range_Stop);
```

```
-- For domains of type enumeration.
type Vals is (Value_1, Value_2, Value_3, ...);
package Fifth is new Dom_1.generate_enum_domain
      (Vals, Domain_Name5, Domain_Type, Null_Indicator);

-- For subtypes of domains of enumeration.
package Sixth is new Dom_1.generate_subenum_domain
      (Domain_Name6, Domain_Based_On, Domain_Type,
          Null_Indicator, Range_Start, Range_Stop);
```

Step 8. After defining all the domains for a single domain package, invoke the generic procedure `Start_Generation`.

```
begin
  Dom_1.start_generation;
end;
```

Now, repeat Steps 5 through 8 for the next domain package.

Step 9. After defining all domain packages, invoke the generic procedure `Generate_Base_Specific`.

```
  Dom_Packs.generate_base_specifc;
```

Step 10. Compile the procedure and correct any indicated errors.

Step 11. Execute the procedure.

All of the Ada code required by Steps 1 through 9 is shown in template form in Table 3.3. When writing a domain definition and generation procedure, use this template as a guide substituting desired names and values for any expressions in italics.

Table 3.3. Domain Definition Template

```
with abstract_domain_generator;                                       -- Step 1

with generator_support, use generator_support;

procedure Domain_View is                                              -- Step 2

  type Pack_Names  is                                                 -- Step 3
      (Package_Name1, Package_Name2, Package_Name3, ...);
  package Dom_Packs is new abstract_domain_generator(Pack_Names);     -- Step 4

  begin
    declare                                                           -- Step 5
      type Doms is (Domain_Name1, Domain_Name2, Domain_Name3, ...);
      package Dom_1 is new                                            -- Step 6
        Dom_Packs.generate_domain_package(Package_Name1, Doms);
    --  For domains of type char, int or smallint.                   -- Step 7
      package First is new Dom_1.generate_int_domain(Domain_Name1,
        Domain_Type, Null_Indicator, Range_Start, Range_Stop);
    --  For subtypes of domains of type char, int or smallint.
      package Second is new Dom_1.generate_subint_domain(Domain_Name2,
        Domain_Based_On, Domain_Type, Null_Indicator, Range_Start,
                         Range_Stop);
    --  For domains of type real, decimal or double_precision.
      package Third is new Dom_1.generate_flt_domain(Domain_Name3,
        Domain_Type, Null_Indicator, Range_Start, Range_Stop);
    --  For subtypes of domains of type real, decimal or
                         double_precision.
      package Fourth is new Dom_1.generate_subflt_domain(Domain_Name4,
        Domain_Based_On, Domain_Type, Null_Indicator, Range_Start,
                         Range_Stop);
    --  For domains of type enumeration.
      type Vals is (Value_1, Value_2, Value_3, ...);
      package Fifth is new Dom_1.generate_enum_domain(Vals, Domain_Name5,
        Domain_Type, Null_Indicator);
    --  For subtypes of domains of enumeration.
      package Sixth is new Dom_1.generate_subenum_domain(Domain_Name6,
        Domain_Based_On, Domain_Type, Null_Indicator, Range_Start,
        Range_Stop);
    -- Additional domains here...

    begin                                                            -- Step 8
      Dom_1.start_generation;
    end;

  -- Additional domain packages here...

  Dom_Packs.generate_base_specifc;                                   -- Step 9

end Domain_View;
```

3.4 DETAILED DESCRIPTION OF DOMAIN GENERICS

Each individual domain definition uses a different generic and a different subset of the the domain generic parameters depending on its SQL type, such as "char" or "int." Each domain to be defined will be of one of the SQL types and, depending on the type, will use one of six possible generics. Domains can be instantiated during Step 7 in any order (regardless of type) with the important exception that any subtype generic must defined only after its parent domain has been defined via a generic instantiation. Descriptions, syntactic and semantic guidelines, and examples of the set of domain generic parameters and each of the six type-based generics are provided in the following paragraphs.

3.4.1 Domain Generic Parameter Set

All of the domain generics use some subset of these seven parameters. This paragraph provides general guidelines, rules, and examples for assigning values to these parameters.

1. `Domain_Name` —The name of the domain being defined. The name should be descriptive of the domain, however, avoid adding such things as "domain," "type," or "not_null" to the names. These suffixes will be added automatically where appropriate.

   Example-> `Parts_Numbers` for a domain describing columns that will contain part numbers or `Employee_Name` for a domain describing columns that contain names.

2. `Domain_Based_On`—The name of the parent domain on which the subtype to be defined is based. Remember that the parent domain must already be defined before defining the subtype.

   Example-> `Part_Numbers_Over_1000` for a subtype based on the Part_Number domain or `Employee_Names_L_To_S` for a subtype based on the Employee_Names domains.

3. `Domain_Type` —The domain's or subtype's SQL type. Subtypes are, of course, always of the same type as the parent domain. The valid SQL types, as defined in the SAME package `Sql_Standard,` are char, int, smallint, real, decimal, double_precision, and enumeration.

   Example-> `char` for an SQL character domain or `real` for an SQL real domain.

4. `Null_Indicator`—This domain's null bearing state and always one of:

   - `null_and_not_null` if the domain has both null bearing and not null types.
   - `contains_null` if the domain is null bearing.
   - `not_null` if the domain is not null bearing.

   A characteristic of the SAME support packages and their treatment of null values is the inability to preform the Ada `assign` procedure on variables of numeric, null bearing (`contains_ null`) domains or subtypes. Therefore, if a numeric domain or subtype needs to be defined as null bearing, but the `assign` procedure will also be needed, use the `null_and_not_null` indicator.

   Example-> `not_null` for a domain describing columns that are constrained to never contain a null value.

5. `Range_Start`—A value representing the beginning of a domain's range. For domains or domain subtypes of type char, int, or smallint, this value is always an numeric integer. For domains or domain subtypes of type real, decimal, or double_precision, this value is always

an numeric real. Domains of type enumeration have no ranges, but for enumeration domain subtypes, this value is one of the parent domain's enumerated values.

Example-> `4` for a domain of type char describing columns will be at least 4 characters long or `5.5` for a domain of type decimal describing columns with a value of at least 5.5.

6. *`Range_Stop`*—A value representing the end of a domain's range.

Example-> `30` for a domain of type char describing columns will be no longer than 30 characters long or `23.6` for a domain of type decimal describing columns with a value no higher than 23.6.

7. *`Vals`*—The name of the enumerated type declaring all of the valid enumeration values for a domain of type enumeration.

Example-> In type color_vals is (red, blue, green), color_vals is the name of the enumerated type, and thus, is the parameter value.

3.4.2 Generics for Char, Int, and Smallint Domains and Subtypes

The `Generate_Int_Domain` generic is used to define domains of type char, int, or smallint. Its generalized syntax is:

```
package A_Package_Name is new
        Second_Level_Generic_Package_Name.generate_int_domain
        (Domain_Name, Domain_Type, Null_Indicator,
                Range_Start, Range_Stop);
```

Example ->

```
package First is new Domain_1.generate_int_domain
        (Employee_Id, int, null_and_not_null, 0, 999);
```

Resulting generated domain definition->

```
type employee_id_not_null is new sql_int_not_null range 0..999;
type employee_id_type is new sql_int;
package employee_id_ops is new
        sql_int_ops(employee_id_type, employee_id_not_null);
```

The `Generate_Subint_Domain` is used to define subtypes of type char, int, or small int. Its generalized syntax is:

```
package A_Package_Name is new
        Second_Level_Generic_Package_Name.generate_subint_domain
        (Domain_Name, Domain_Based_On, Domain_Type,
        Null_Indicator, Range_Start, Range_Stop);
```

Example ->

```
package Second is new domain_1.generate_sub_domain
      (Id_Over_50, Employee_Id, int, null_and_not_null, 50, 999);
```

Resulting generated domain definition->

```
subtype id_over_50_not_null is employee_id_not_null
     range 51..100;
subtype id_over_50_type is new employee_id_type;
package id_over_50_ops is new
     sql_int_ops(id_over_50_type, id_over_50_not_null);
```

3.4.3 Generics for Real, Decimal and Double_Precision Domains and Subtypes

The `Generate_Flt_Domain` is used for domains of type real, decimal, or double_precision. Its generalized syntax is:

```
package A_Package_Name is new
        Second_Level_Generic_Package_Name.generate_flt_domain
                (Domain_Name, Domain_Type, Null_Indicator,
                 Range_Start, Range_Stop);
```

Example ->

```
package Third is new Domain_1.generate_flt_domain
      (Weekly_Earnings, decimal, not_null, 0.00, 1500.00);
```

Resulting generated domain definition->

```
type weekly_earnings_not_null is new sql_real_not_null
     range 0.00..1500.00;
```

The `Generate_Subflt_Domain` is used for subtypes of type real, decimal, or double_precision. Its generalized syntax is:

```
package A_Package_Name is new
       Second_Level_Generic_Package_Name.generate_subflt_domain
       (Domain_Name, Domain_Based_On, Domain_Type,
        Null_Indicator, Range_Start, Range_Stop);
```

Example ->

```
package Fourth is new Domain_1.generate_subflt_domain
       (Low_Earnings, decimal, not_null, 0.00, 500.00);
```

Resulting generated domain definition->

```
subtype low_earnings_not_null is weekly_earnings_not_null
        range 0.00..500.00;
```

3.4.4 Generics for Enumeration Domains and Subtypes

The `Generate_Enum_Domain` is used for domains of type enumeration. Its generalized syntax is:

```
type Vals is (Value_1, Value_2, Value_3, ...);
package A_Package_Name is new
        Second_Level_Generic_Package_Name.generate_enum_domain
        (Vals, Domain_Name, Domain_Type, Null_Indicator);
```

Example ->

```
type Color_Vals is (red, white, blue);
package Fifth is new domain_1.generate_enum_domain
             (Color_Vals, Colors, enumeration, null_and_not_null);
```

Resulting generated domain definition->

```
type colors_not_null is (red, white, blue);
package colors_ops is new
        sql_enumeration_pkg(colors_not_null);
type colors_type is new colors_ops.sql_enumeration;
```

The `Generate_Subint_Domain` is used for subtypes of type enumeration. Its generalized syntax is:

```
package A_Package_Name is new
        Second_Level_Generic_Package_Name.generate_subenum_domain
        (Domain_Name, Domain_Based_On, Domain_Type, Null_Indicator,
        "Range_Start", "Range_Stop");
```

Note: The range values for enumeration subtypes must in double quotes.

Example ->

```
package Sixth is new domain_1.generate_enum_domain
          (Colors_Not_Red, Colors, enumeration, null_and_not_null,
          "White", "Blue");
```

Resulting generated domain definition->

```
subtype colors_not_red_not_null is colors_not_null
        range white..blue;
package colors_not_red_ops is new
        sql_enumeration_pkg(colors_not_red_not_null);
type colors_not_red_type is new
        colors_not_red_ops.sql_enumeration;
```

3.5 EXAMPLE DOMAIN DEFINITION PROCEDURE

Given the parameters used in the SQL module shown in Table 3.2, seven domains can be defined. The code in Table 3.4 can be used to generate those domain definitions and was written using the template illustrated in Table 3.3 as a guideline. In essence, the code was written by fill-in-the-blank technique.

Table 3.4. Example Domain Definition Procedure Dixie_Domain_View

```
-- This procedure is used to define and generate abstract domains
-- corresponding to column types in the dixie_db data base.

with abstract_domain_generator;                              -- Step 1
with generator_support; use generator_support;

procedure dixie_domain_view is                               -- Step 2

  type package_names is                                      -- Step 3
          (dixie_employee_def_pkg, dixie_salary_def_pkg);

  package dom_packs is new abstract_domain_generator         -- Step 4
          (package_names);
begin
    declare                                                  -- Step 5
      type doms is (employee_id, name, address, weight);

      package domain_1 is new dom_packs.generate_domain_package  -- Step 6
            (dixie_employee_def_pkg, doms);

        package first is new domain_1.generate_int_domain    -- Step 7
            (employee_id, char, not_null, 1, 5);
        package second is new domain_1.generate_int_domain
            (name, char, not_null, 1, 10);
        package third is new domain_1.generate_int_domain
            (address, char, not_null, 1, 30);
        package four is new domain_1.generate_flt_domain
            (weight, real, not_null, 0.0, 300.0);

    begin                                                    -- Step 8
        domain_1.start_generation;
    end;

    -- Repeated Steps 5 - 8 below for the second domain package
    declare
      type doms is (title, points, salary);

      package domain_2 is new dom_packs.generate_domain_package
          (dixie_salary_def_pkg, doms);
        package five is new domain_2.generate_int_domain
          (title, char, contains_null, 1, 15);
        package six is new domain_2.generate_int_domain
            (points, int, null_and_not_null, 0, 99);
        package seven is new domain_2.generate_flt_domain
            (salary, decimal, not_null, 0.00, 99000.00);

    begin
      domain_2.start_generation;
    end;

    dom_packs.generate_base_specific;                       -- Step 9

end dixie_domain_view;
```

3.6 RESULTS OF OPERATION

The end results of the `Dixie_Domain_View` procedure are the generated domain packages, `Dixie_Employee_Def_Pkg` and `Dixie_Salary_Def_Pkg`, and the file `Base_Specific_Domains`. Table 3.5 illustrates these generated products.

Table 3.5. Generated Products

Dixie_Employee_Def_Pkg - Specification

```
with sql_char_pkg;
use sql_char_pkg;
with sql_real_pkg;
use sql_real_pkg;
package dixie_employee_def_pkg is

  type employee_idnn_base is new sql_char_pkg.sql_char_not_null;
  subtype employee_id_not_null is employee_idnn_base (1..5);

  type namenn_base is new sql_char_pkg.sql_char_not_null;
  subtype name_not_null is namenn_base (1..10);

  type addressnn_base is new sql_char_pkg.sql_char_not_null;
  subtype address_not_null is addressnn_base (1..30);

  type weight_not_null is new sql_real_not_null
```

```
        range 0.00000..300.00000;

end dixie_employee_def_pkg;
```

Dixie_Salary_Def_Pkg - Specification

```
with sql_char_pkg;
use sql_char_pkg;
with sql_int_pkg;
use sql_int_pkg;
with sql_real_pkg;
use sql_real_pkg;
package dixie_salary_def_pkg is
  type title_base is new sql_char_pkg.sql_char;
  subtype title_type is title_base;
  type points_not_null is new sql_int_not_null
        range 0..99;
  type points_type is new sql_int;
  package points_ops is new
        sql_int_ops(points_type, points_not_null);

  type salary_not_null is new sql_real_not_null
        range 0.00000..99000.00000;

end dixie_salary_def_pkg;
```

Base_Specific_Domains - Specification

```
with generator_support;
use generator_support;
package base_specific_domains is

  type base_specific_domain_types is (employee_id_not_null, name_not_null,
        address_not_null, weight_not_null, title_type, points_not_null,
        points_type, salary_not_null, null_domain_type);

  type corresponding_concrete_types is (employee_id_not_null_char_5,
        name_not_null_char_10, address_not_null_char_30,
        weight_not_null_real, title_type_char_15, points_not_null_int,
        points_type_int, salary_not_null_real);
  type valid_domain_names is (dixie_employee_def_pkg, dixie_salary_def_pkg);

  type ops_packages is (points_ops);

  longest_enum_value : constant integer :=  0;

  type domain_package_array is array (positive range <>) of
       valid_domain_names;

end base_specific_domains;
```

These packages are to be compiled with no further coding. Once compiled, the domain packages are made available to the application programmer, and the `Base_Specific_ Domains` package is ready for input during interface definition and generation.

3.7 SUMMARY

In summary, the following tasks must be performed during domain definition and generation.

Table 3.6. Tasks to be Performed During Domain Definition and Generation

Tasks	Outcome
1. Create a working directory containing the compiled ADA/SQL binding tool.	
2. Using the SQL Module, determine the necessary domain definitions.	
3. Write an Ada procedure using `Abstract_Domain_Generator` to describe the domains. Use the template provided in Table 3.3.	
4. Compile and execute the procedure.	Two or more files: the domain packages and the package `Base_Specific_Domains`.
5. Compile the domain packages.	Semantically correct SAME domains to support the application.
6. Compile the package `Base_ Specific_Domains` for later input.	Semantically correct package describing the valid domains.

Section 4
Defining and Generating the Interface

4.1 GENERAL PREPARATION

After compiling the domain packages and the file Base_Specific_Domains, the interface programmer compiles the packages Concrete_Package and Abstract_Interface_ Generator. Both of these packages import the Base_Specific_Domains specification, and because the information in Base_Specific_Domains is specific to each set of domain packages, compilation of Abstract_Interface_Generator and Concrete_Package is required with every new Base_Specific_Domains specification. After these compilations, the interface programmer is ready to begin interface definition.

The programmer must refer to the SQL module to determine what Ada procedures will be needed in the abstract interface to represent the SQL data manipulation statements in the concrete module. There is a one-to-one mapping between procedures in the concrete module and procedures in the abstract interface specification. Central to interface definition is the mapping of each of the SQL statements in the SQL module to one of these Ada procedures. See *Guidelines for the Use of the SAME* [1] for further explanation. When the interface programmer writes the Ada procedure that generates the abstract interface, these Ada procedures and parameters will be described to generate the actual interface procedures and parameters. Table 4.1 shows the SQL statement types, their matching Ada procedure, and the parameters and modes for each kind of Ada procedure.

At this time, the interface programmer is also responsible for providing access to DBMS error/status calls from the application program. A feature of the SQL language provides that the execution of every SQL statement returns a status code, SQLCODE, indicating statement execution success or type of failure. Though there is a core of ANSI standard SQLCODE values, some additional values are DBMS implementation dependent. The programmer determines what types of errors are expected from the data base in the form of SQLCODE during application program execution. Each expected error and its SQLCODE should be mapped to an enumeration value representing that error. For example, if a fetch procedure is expected, occasionally, to fail to find row and return a SQLCODE value of 100, the enumeration value row_not_found could represent this error. An exception to this is the SQLCODE value for transaction success. This SQLCODE value should not be represented because it is automatically provided by the interface. SAME also provides error processing and recovery capabilities through two SAME packages, Sql_Communications_Pkg and Sql_Database_Error_Pkg. These packages can be modified to meet the error processing requirements of the application. Instructions for modifying the packages for the Digital RDB DBMS are found in Section 6.

4.2 INPUTS FOR INTERFACE DEFINITION AND GENERATION

Based on SQL statements coded in the SQL module, the interface programmer has determined the actual Ada procedures, associated row records or individual parameters, and error processing needed to generate the interface. At this point, a unique name should be determined for each procedure, row record and individual parameter that will be part of the interface. Next the interface programmer writes an Ada procedure, *Interface_View,* to create the abstract

Table 4.1. SQL Statements and Corresponding Ada Procedure Kinds

SQL Statement Type	Ada Procedure Kind	Ada Parameter Kind	Mode
CLOSE	`close`	none	
COMMIT	`commit`	none	
DELETE (positioned)	`delete_positioned`	none	
DELETE (searched)	`delete_searched`	individual parameters	in
FETCH	`fetch`	row record*	in out
INSERT VALUES	`insert_values`	row record	in
INSERT (subquery)	`insert_subquery`	individual parameters	in
OPEN (cursor)	`open`	individual parameters	in
ROLLBACK	`rollback`	none	
SELECT	`selec**`	row record	in out
		individual parameters	in
UPDATE (positioned)	`update_positioned`	individual parameters	in
UPDATE (searched)	`update_searched`	individual parameters	in

* Row record is an Ada record in which each record component represents one of the columns of an SQL row record. A unique row record is not required for each procedure (i.e. more than one procedure can take the same row record as a parameter).
** "Select" is a reserved word in Ada.

interface. This procedure instantiates several layers of nested generics contained in `Abstract_Interface_ Generator`. When executed, it produces four text files:

- The Ada specification and body (each a file) of the `Abstract_Interface`
- The Ada specification of the `Concrete_Interface`
- A text file, *`DBMS_Specific`*, that contains information needed to modify the concrete interface specification if any DBMS-required changes are needed.

The following paragraphs provide the steps and guidelines for writing the Ada procedure, *`Interface_View`*, that will generate these files.

4.3 STEPS AND GUIDELINES FOR PROCEDURE *`Interface_View`*

To define the interface, the interface programmer always follows the same basic steps. These steps take the form of writing an Ada procedure that instantiates each of the three levels of nested generic within the `Abstract_Interface_Generator` generic package. This does not mean that every procedure or generic package is used but that at least one generic package in each level of the nesting must be instantiated.

The following numbered steps are described in the order in which the interface programmer writes the Ada code. Each step includes example code that implements the step and a complete procedure template, which also shows the steps, is shown in Table 4.2. Note that italics indicate user-supplied names and values. Names can be any valid Ada name but must be used consistently throughout the procedure.

Step 1. Import 3 packages:

`Base_Specific_Domains` => contains declarations of all valid domain packages and domains. This package must always be imported.

`Abstract_Interface_Generator` => contains the generics that when instantiated generate the interface packages. This package must always be imported.

`Generator_Support` => contains common declarations (mostly enumerated types and constants). This package must always be imported.

```
with base_specific_domains; use base_specific_domains;
with abstract_interface_generator;
with generator_support; use generator_support;
```

Step 2. Create an Ada main procedure.

```
procedure Interface_View is
  begin
end Interface_View;
```

Step 3. Within this procedure declare an enumerated type representing all valid names of the row records needed.

Note: Duplicate record names are not allowed.

```
type Record_Names  is
       (Record_Name1, Record_Name2, Record_Name3, ...);
```

Step 4. Declare an enumerated type representing all valid names of the procedures needed.

Note: Duplicate procedure names are not allowed.

```
type Procedure_Names  is
       (Procedure_Name1, Procedure_Name2, ...);
```

Step 5. Declare an enumerated type representing all valid names of any errors expected by the user's application. These names will be mapped in the next step to specific SQLCODE values.

Note: Each SQLCODE value expected requires its own named error, and duplicate error names are not allowed. Note also that the SQLCODE value for transaction success is not to be represented because it is automatically provided.

```
type Error_Names  is
     (Error_Name1, Error_Name2, Error_Name3, ...);
```

Step 6. Instantiate the outermost generic package.

`Abstract_Interface_Generator` with the following eight parameters:

- *Concrete_Name_String*—a string representing the desired name of the concrete interface specification. This name will also appear as the filename.
- *Abstract_Name_String*—a string representing the desired name of the abstract interface specification and body. This name will also appear as the filename (with appropriate extensions) for both files.
- *Dbms_Specific_String*—a string representing the desired name of the dbms_specific text file. This name will appear as the filename (with the extension "`.txt`".
- `(1=>` *Domain_Package1*, `2=>` *Domain_Package2*, `3=>....)`—an array of the domain package names that will be used by this interface. These are the packages (generated by the steps in Section 4) containing the domain declarations. From now on, only the domains in these packages will be recognized as valid domains during elaboration of subsequent generics.
- *Record_Names*—the previously enumerated type representing all valid row record names. Only these record names will be recognized as valid during elaboration of subsequent generics.
- *Procedure_Names*—the previously enumerated type representing all valid procedure names. Only these procedure names will be recognized as valid during elaboration of subsequent generics.
- *Error_Names*—the previously enumerated type representing all valid error names. Only these error names will be recognized as valid during elaboration of subsequent generics.
- `(1=>` *Error_Value1*, `2=>` *Error_Value2*, `3=>....)`—an array of integer values representing the SQLCODE values expected from the data base and corresponding to the errors named previously. The error values must be enumerated in exactly the same order as their corresponding errors were enumerated.

Example:

```
package Int_Packs is new abstract_interface_generator
  ("Concrete_Name_String","Abstract_Name_String",
  "Dbms_Specific_String", (1=> Domain_Package1, 2=>
  Domain_Package2, 3=>....),Record_Names, Procedure_Names,
  Error_Names, (1=> Error_Value1, 2=> Error_Value2, 3=>....));
```

Step 7. Steps 7 through 10 are repeated once for each row record, *Record_Name1*, *Record_Name2*, etc., enumerated in Step 3. For each row record, declare an enumerated type representing the name of each component of the row record. Record component names should be descriptive of their corresponding SQL column.

Note: If this record is to be used to track string truncation, the components must be named exactly the same as the corresponding component in the row record.

```
declare
type Rec_Components is
      (Component_Name1, Component_Name2, ...);
```

Step 8. Instantiate a second-level generic package `Record_Generator` with

- the type, *`Rec_Components`*, representing the components enumerated in the previous step
- the name of the record from the enumerated list in Step 3
- a boolean value indicating that the record is used for returning SQL indicator values to the application (`true`), or if it is used to store or fetch an SQL row (`false`).

From now on, only these component names will be recognized as valid component names for this row record during elaboration of subsequent generics.

```
package Rec_1 is new Int_Packs.record_generator
      (Rec_Components, Record_Name1, true_or_false_value);
```

Step 9. Step 9 is repeated once for each component, *`Component_Name1`*, *`Component_Name2`*, etc., in a particular record and in the same order as the components are enumerated in Step 7. Instantiate a third-level generic package, `Component_Generator`, with two parameters. The first parameter is the record component's name from the set of names enumerated in Step 7. The second parameter is the component's domain type as declared in one of the domains packages of Step 6. If this record is to be used for tracking string truncation, however, the second parameter is not given.

```
package Com_11 is new Rec_1.component_generator
      (Component_Name1, Component_Domain);
```

Repeat this step for the next component of this row record.

Step 10. After defining all row records, invoke the procedure `Generate_Record`.

```
begin
  Rec_1.generate_record;
end;
```

Now, repeat Steps 7 through 10 for the next row record.

Step 11. Step 11 is repeated once for each procedure, *`Procedure_Name1`*, *`Procedure_Name2`*, etc., enumerated in Step 4. For each procedure, instantiate a second-level (and possibly third-level) generic package with parameters that describe the procedure and invoke the generic procedure `Generate_Procedure`. Which generic to instantiate and what parameters to use depend on whether the procedure itself has parameters. An example of each one of the two possible generic instantiations is shown here, however, more detailed descriptions of each generic, along with parameter explanations, are found in the next section.

```
--  For procedures without parameters.
declare
  package Procedure_1 is new
  Int_Packs.procedure_without_parameters_generator
  (Procedure_Name, Procedure_Type, "Concrete_Proc_Name");
begin
  Procedure_1.generate_procedure;
end;

--  For procedures with parameters.
declare
  type Abstract_Params  is
      (Abs_Parameter_Name1, Abs_Parameter_Name2,
        ...);
  type Concrete_Params  is
      (Conc_Parameter_Name1, Conc_Parameter_Name2, ...);

  package Procedure_2 is new
      Int_Packs.procedure_with_parameters_generator
      (Procedure_Name, Abstract_Params,
      Procedure_Type,
      "Concrete_Proc_Name",  Concrete_Params
      (1=>Error_1, 2=> Error_2, ...));
      package Param_1 is new
          Procedure_2.params_of_domain_type_generator
            (Abs_Parameter_Name, Parameter_Type);
      package Param_2 is new
          Procedure_2.params_of_record_type_generator
            (Abs_Parameter_Name, Record_Type);
      package Param_3 is new
          Procedure_2.params_of_boolean_type_generator
            (Abs_Parameter_Name);
      package Param_4 is new
          Procedure_2.params_of_error_conditions_generator
            (Abs_Parameter_Name);

begin
   Procedure_2.generate_procedure;
end;
```

Repeat this step for the next procedure.

Step 12. After defining all row records and procedures invoke the generic procedure `Generate_Interface`.

```
Int_Packs.generate_interface;
```

Step 13. Compile the procedure and correct any indicated errors.

Step 14. Execute the procedure.

All of the Ada code required by Steps 1 through 12 is shown in template form in Table 4.2. When writing an interface definition and generation procedure, use this template as a guide, substituting desired names and values for any expressions in italics.

Table 4.2. Interface Definition Template

```
with base_specific_domains; use base_specific_domains;                -- Step 1
with abstract_interface_generator;
with generator_support; use generator_support;

procedure Interface_View is                                           -- Step 2

  type Record_Names  is                                               -- Step 3
      (Record_Name1, Record_Name2, Record_Name3, ...);

  type Procedure_Names  is                                            -- Step 4
      (Procedure_Name1, Procedure_Name2, Procedure_Name3, ...);

  type Error_Names  is                                                -- Step 5
      (Error_Name1, Error_Name2, Error_Name3, ...);

  package Int_Packs is new abstract_interface_generator               -- Step 6
          ("Concrete_Name_String ", "Abstract_Name_String",
          "Dbms_Specifc_String",
          (1=> Domain_Package1, 2=> Domain_Package2, 3=>....),
          Record_Names, Procedure_Names, Error_Names,
          (1=> Error_Value1, 2=> Error_Value2, 3=>....));

begin

    declare                                                           -- Step 7
      type Rec_Components is
          (Component_Name1, Component_Name2, Component_Name3, ...);

      package Rec_1 is new Int_Packs.record_generator                 -- Step 8
        (Rec_Components, Record_Name1, Rec_Indicator);

        package Com_11 is new Rec_1.component_generator               -- Step 9
          (Component_Name1, Component_Domain);
        package Com_12 is new Rec_1.component_generator
          (Component_Name2, Component_Domain);
        package Com_13 is new Rec_1.component_generator
          (Component_Name3, Component_Domain);
      -- Additional record components here...

  begin                                                         -- Step 10
      Rec_1.generate_record;
  end;
    -- Additional row records here...

    --  For procedures without parameters.                      -- Step 11
  declare
    package Procedure_1 is new
      Int_Packs.procedure_without_parameters_generator
        (Procedure_Name, Procedure_Type, "Concrete_Proc_Name");
  begin
    Procedure_1.generate_procedure;
  end;

    --  For procedures with parameters.
  declare
    type Abstract_Params  is
        (Abs_Parameter_Name1, Abs_Parameter_Name2, ...);
    type Concrete_Params  is
        (Conc_Parameter_Name1, Conc_Parameter_Name2, ...);

    package Procedure_2 is new
      Int_Packs.procedure_with_parameters_generator
        (Procedure_Name, Abstract_Params, Procedure_Type,
        "Concrete_Proc_Name", Concrete_Params,
        (1=>Error_1, 2=> Error_2, ...));

      package Param_1 is new
        Procedure_2.params_of_domain_type_generator
          (Abs_Parameter_Name, Parameter_Type);
      package Param_2 is new
        Procedure_2.params_of_record_type_generator
          (Abs_Parameter_Name, Record_Type);
      package Param_3 is new
        Procedure_2.params_of_boolean_type_generator
          (Abs_Parameter_Name);
      package Param_4 is new
        Procedure_2.params_of_error_conditions_generator
          (Abs_Parameter_Name);
        -- Additional parameters here...

    begin
        Procedure_2.generate_procedure;
    end;
      -- Additional procedures here...

  Int_Packs.generate_interface;                              -- Step 12

end Interface_View;
```

4.4 DETAILED DESCRIPTION OF PROCEDURE GENERICS

As shown previously, there are two types of procedure definition generics, one for procedures with parameters and one for procedures without parameters. An additional third-level generic is used for procedures with parameters to define each of the procedure parameters. Refer to Table 4.1 for a listing of procedures that take parameters. Descriptions, syntactic and semantic guidelines, and examples of the two types of procedure generics are provided below.

4.4.1 Procedures without Parameters

The generic `Procedure_Without_Parameters_Generator` is used to generate the Ada procedures corresponding to the SQL statements that do not take parameters, such as procedure `commit`. This generic is instantiated with three parameters:

- *Procedure_Name*—a procedure name from the previously enumerated list of procedure names represented by the type *Procedure_Names* in Paragraph 4.3, Step 4.
- *Procedure_Type*—the type of procedure, such as `update_positioned`, or `close`. Table 4.1 shows the 12 possible procedure types.
- *Concrete_Proc_Name*—a string in double quotes representing the name of the corresponding procedure in the SQL module. The abstract procedure name and the concrete procedure name do not have to be the same.

Following each instantiation of this generic, invoke the generic procedure Generate_Procedure. The generalized syntax is:

```
declare
package Procedure_1 is new
         Int_Packs.procedure_without_parameters_generator
         (Procedure_Name, Procedure_Type, "Concrete_Proc_Name");
begin
  Procedure_1.generate_procedure;
end;
```

Example ->

```
declare
package Proc1 is new
         Int_Packs.procedure_without_parameters_generator
         (Start_Over, Rollback, "Rollback_Transaction");
```

```
begin
  Proc1.generate_procedure;
end;
```

4.4.2 Procedures with Parameters

The `Procedure_With_Parameters_Generator` is used to generate the Ada procedures corresponding to the SQL statements that do take parameters, such as procedure `fetch`. When defining the Ada procedures with parameters, it is not necessary to indicate parameter mode. All modes are automatically built into the abstract interface based on procedure kind (see Table 4.1). Defining procedures with parameters requires the following steps:

Step 1. Declare an enumerated type representing the parameters for this specific abstract procedure.

> **Note: Indicator parameters are not declared here since they are generated automatically.**

Parameters can be:

- A single parameter of domain type where that domain type is one of the domain types in the domain packages.
- A record parameter of record type where that record type is one of the record types declared in Paragraph 4.3, Step 3 for a row record or string indicator record.

Either:

- A parameter of boolean type used to return a true or false value as SQL statement result information to the application. This boolean result parameter can either be mapped to one (and only one) of the expected errors or to no expected errors. In the first case, the result parameter will be returned automatically as false to the application if the Ada procedure's corresponding SQL statement returns the SQLCODE for that one expected error. In the second case, the result parameter will be returned automatically as false to the application if the Ada procedure's corresponding SQL statement results in an SQLCODE value not equal to 0 (i.e., if the statement fails for whatever reason). In this instance, the application is only interested in pass/fail information, not the reason the statement failed.

Or:

- A parameter name to return an expected error (from the set of errors declared in Paragraph 4.3, Step 5). This error condition result parameter will be used to return the applicable error based on SQLCODE returned from the SQL statement. This parameter will be associated with a specific subset of expected errors in the following Step 3.

> **Note: Declaring both a boolean result parameter and an error condition result parameter is not allowed. Duplicate parameter names are also not allowed.**

The generalized syntax is:

```
declare
type Abstract_Params  is
        (Abs_Parameter_Name1, Abs_Parameter_Name2, ...);
```

Example ->
```
declare
type Proc2_Abs_Params is
       (Employee_Id, Personnel_Record, Row_Not_Found);
```

Step 2. Next, declare an enumerated type representing the parameters to the corresponding concrete procedure in the SQL module. Parameters must be listed in the same order as those in the SQL module procedure. Parameters can be:

- A single domain type parameter from the list of abstract parameters.
- A record component type parameter. If a record was one of the parameters to the abstract interface, then every component of that record must be represented by a separate component type parameter in this step. The record itself may not be used, only its separate components.

Either:

- The boolean type result parameter from the list of parameters represented by *Abstract_Params*.

Or:

- The error condition type result parameter from the list of parameters represented by *Abstract_Params*.

Note: Duplicate parameter names are not allowed.

The generalized syntax is:
```
type Concrete_Params is
       (Conc_Parameter_Name1, Conc_Parameter_Name2, ...);
```

Example ->
```
type Proc2_Conc_Params is
       (Employee_Id, Name, Address, Id_Number,
        Row_Not_Found);

-- Name, Address, Id_Number are the components of
      Personnel_Record.
```

Step 3. Instantiate the `Procedure_With_Parameters_Generator` with six parameters:

- *Procedure_Name*—the procedure name from the previously enumerated list of procedure names represented by the type *Procedure_Names* in Paragraph 4.3, Step 4.
- *Abstract_Params*—the enumerated type representing the procedure's abstract parameters declared in step 1 above.
- *Procedure_Type*—the type of procedure, such as `update_positioned`, or `close`. Table 4.1 shows the 12 possible procedure types.

- *Concrete_Proc_Name*—a string in double quotes representing the name of the corresponding procedure in the SQL module. The abstract procedure name and the concrete procedure name do not have to be the same.

- *Concrete_Params*—the enumerated type representing the procedure's concrete parameters declared in Step 2.

- (1=> *Error_1*, 2=> *Error_2*, ...)—any one of a number of errors expected from the data base. Each error must be of the valid errors declared in Paragraph 4.3, step 5. The error condition type result parameter in step 1 represents this error set. If a boolean type parameter was specified in the *Abstract_Params* declaration of step 1, then either give only one expected error or give no errors.

The generalized syntax is:

```
package Procedure_2 is new
   Int_Packs.procedure_with_parameters_generator
   (Procedure_Name, Abstract_Params, Procedure_Type,
   "Concrete_Proc_Name", Concrete_Params,
   (1=>Error_1, 2=> Error_2, ...));
```

Example when *Abstract_Params* has an error condition type result parameter->

```
package Proc2 is new
   Int_Packs.procedure_with_parameters_generator
   (Fetch_On_Id, Abstract_Pars, fetch, "Id_Fetch",
    Concrete_Pars, (1=>row_not_found, 2=> bad_record));
```

Example when *Abstract_Params* has a boolean type result parameter (one error)->

```
package Proc2 is new
   Int_Packs.procedure_with_parameters_generator
   (Update_On_Id, Abstract_Pars, update, "Id_Update",
    Concrete_Pars, (1=>row_not_found));
```

Example when *Abstract_Params* has a boolean type result parameter (no errors)->

```
package Proc2 is new
   Int_Packs.procedure_with_parameters_generator
   (Delete_On_Id, Abstract_Pars, delete, "Id_Delete",
    Concrete_Pars);
```

Step 4. Next, instantiate one of four parameter generics for each parameter of the abstract procedure represented by type *Abstract_Params*. Which parameter generic used depends on parameter type:

- For domain type parameters—instantiate the generic Params_Of_**Domain_Type**_Generator with two parameters. The first parameter, *Abs_Parameter_Name*, is the name of the parameter from the list represented by *Abstract_Params*. The second parameter, *Parameter_Type*, is the abstract domain type of this parameter.

  The generalized syntax is:
  package *Param_1* is new
  Procedure_2.params_of_domain_type_generator
  (*Abs_Parameter_Name*, *Parameter_Type*);
  Example->

```
package First_Parameter is new
        Proc2.params_of_domain_type_generator
              (Employee_Id, Id_Numbers_Not_Null);
```

- For record type parameters—instantiate the generic `Params_Of_`**`Record_`** **`Type`**`_Generator` with two parameters. The first parameter, *`Abs_Parameter_Name`*, is the name of the record from the list represented by *`Abstract_Params`*. The second parameter, *`Record_Type`*, is the record type of this parameter. The generalized syntax is:

```
package Param_2 is new
        Procedure_2.params_of_record_type_generator
              (Abs_Parameter_Name, Record_Type);

        Example->
        package Second_Parameter is new
        Proc2.params_of_record_type_generator
              (Personnel_Record, Employee_Rec_Type);
```

- For boolean result parameters—instantiate the generic Params_Of_**Boolean_ Type**_Generator with one parameter, *Abs_Parameter_Name*, the name of the parameter from the list represented by *Abstract_Params*. The generalized syntax is:

```
package Param_3 is new
        Procedure_2.params_of_boolean_type_generator
              (Abs_Parameter_Name);

Example->
package Third_Parameter is new
        Proc2.params_of_boolean_type_generator
              (Employee_Exists);
```

- For error condition result parameters—instantiate the generic Params_Of_**Error_ Conditions**_Generator with one parameter, *Abs_Parameter_Name*, the name of the parameter from the list represented by *Abstract_Params*.

```
package Param_4 is new
        Procedure_2.params_of_error_conditions_generator
              (Abs_Parameter_Name);

Example->
package Fourth_Parameter is new
    Proc2.params_of_error_conditions_generator
          (Error_Result);
```

Step 5. Following each complete procedure and parameters definition, invoke the generic procedure Generate_Procedure. This generalized syntax is:

```
begin
  Procedure_2.generate_procedure;
end;
```

Example ->
```
begin
  Proc2.generate_procedure;
end;
```

4.5 EXAMPLE INTERFACE DEFINITION PROCEDURE

Given the procedures and parameters used in the SQL module shown in Table 3.1, eight procedures and two row records require definition. The code in Table 4.3 can be used to generate an interface for those procedures and was written using the template illustrated in Table 4.2 as a guideline. As with the domain definition procedure, the code was written by fill-in-the-blank technique.

Table 4.3. Example Interface Definition Procedure Dixie_Interface_View

```
— This procedure is used to define and generate the abstract interface
— corresponding to the SQL statements in the SQL module dixie_conc_interface.

with base_specific_domains; use base_specific_domains;          -- Step 1
with abstract_interface_generator;
with generator_support; use generator_support;

procedure dixie_interface_view is                               -- Step 2

  type the_records is (emp_rec, sal_rec);                       -- Step 3

  type the_procedures is (startup, open_emp_cursor,             -- Step 4
                          emp_fetch, new_sal_row, add_year,
                          keep_it, trash_it, close_emp_cursor);

  type the_errors is (row_not_found, cursor_already_open,       -- Step 5
                          no_duplicates, cursor_not_open);

  package make_interface is new abstract_interface_generator    -- Step 6
          ("dixie_conc_interface", "dixie_abs_interface",
           "dixie_dbms_specific",
           (1=> dixie_employee_def_pkg, 2=> dixie_salary_def_pkg),
           the_records, the_procedures, the_errors,
           (1=> 100, 2=> 1001, 3=> -803, 4=> -501));
```

```
begin
    declare                                                            -- Step 7
      type rec_comps1 is (emp_id, emp_lastname, emp_firstname,
                          emp_address, emp_weight);

      package rec1 is new make_interface.record_generator              -- Step 8
                          (rec_comps1, emp_rec, false);

        package comp_11 is new rec1.component_generator                -- Step 9
             (emp_id, employee_id_not_null);
        package comp_12 is new rec1.component_generator
             (emp_lastname, name_not_null);
        package comp_13 is new rec1.component_generator
             (emp_firstname, name_not_null);
        package comp_14 is new rec1.component_generator
             (emp_address, address_not_null);
        package comp_15 is new rec1.component_generator
             (emp_weight, weight_not_null);
    begin                                                              -- Step 10
        rec1.generate_record;
    end;

    -- Repeated Steps 7 - 10 below for the second record
    declare
      type rec_comps2 is (new_emp_id, new_emp_sal, new_emp_pts);

      package rec2 is new make_interface.record_generator
                          (rec_comps2, sal_rec, false);

        package comp_21 is new rec2.component_generator
                          (new_emp_id, employee_id_not_null);
        package comp_22 is new rec2.component_generator
                          (new_emp_sal, salary_not_null);
        package comp_23 is new rec2.component_generator
                          (new_emp_pts, points_type);
    begin
        rec2.generate_record;
    end;
```

```
declare                                                        -- Step 11
  package proc1 is new
        make_interface.procedure_without_parameters_generator
                       (startup, commit, "set_transaction");
  -- The set transaction SQL procedure is only used with the RDB DBMS.
  -- Therefore, it is not one of the standard SQL statement types and has
  -- no corresponding ada procedure kind.  "commit" can be used though
  -- because it too is a kind of procedure without parameters like set
  -- transaction.
begin
    proc1.generate_procedure;
end;

-- Repeated step 11 for seven additional procedures
declare
  package proc2 is new
        make_interface.procedure_without_parameters_generator
                       (keep_it, commit, "commit_transaction");
begin
    proc2.generate_procedure;
end;

declare
  package proc3 is new
        make_interface.procedure_without_parameters_generator
                       (close_emp_cursor, close, "close_emp_cursor");
begin
    proc3.generate_procedure;
end;

declare
  package proc4 is new
        make_interface.procedure_without_parameters_generator
                       (trash_it, rollback, "rollback_transaction");
begin
    proc4.generate_procedure;
end;
```

```
declare
  type abs_params5 is (in_emp_id, result);
  type con_params5 is (in_emp_id, result);

  package proc5 is new make_interface.procedure_without_parameters_generator
        (open_emp_cursor, abs_params5, open, "open_employees_cursor",
          con_params5, (1=> cursor_already_open));

    package param51 is new proc5.params_of_domain_type_generator
        (in_emp_id, employee_id_not_null);
    package param52 is new proc5.params_of_error_conditions_generator
        (result);
begin
    proc5.generate_procedure;
end;

declare
  type abs_params6 is (fetch_rec, fetch_done);
  type con_params6 is ( emp_id, emp_lastname,
                        emp_firstname, emp_address, emp_weight, fetch_done);

  package proc6 is new make_interface.procedure_with_parameters_generator
          (emp_fetch, abs_params6, fetch, "fetch_employees",
          con_params6, (1=> row_not_found, 2=> cursor_not_open));
    package param61 is new proc6.params_of_record_type_generator
                                          (fetch_rec, emp_rec);
    package param62 is new proc6.params_of_error_conditions_generator
                                          (fetch_done);
begin
  proc6.generate_procedure;
end;

declare
  type abs_params7 is (insert_rec, insert_done);
  type con_params7 is (new_emp_id, new_emp_sal, new_emp_pts, insert_done);

  package proc7 is new make_interface.procedure_with_parameters_generator
       (new_sal_row, abs_params7, insert_values, "insert_sal_hist_row",
        con_params7, (1=> no_duplicates));
        package param71 is new proc7.params_of_record_type_generator
          (insert_rec, sal_rec);
        package param72 is new proc7.params_of_error_conditions_generator
          (insert_done);
    begin
        proc7.generate_procedure;
    end;

    declare
      type abs_params8 is (in_emp_id, change_done);
      type con_params8 is (in_emp_id, change_done);

      package proc8 is new make_interface.procedure_with_parameters_generator
            (add_year, abs_params8, update_searched, "update_job_hist",
             con_params8);

        package param81 is new proc8.params_of_domain_type_generator
            (in_emp_id, employee_id_not_null);
        package param82 is new proc8.params_of_boolean_type_generator
            (change_done);
    begin
        proc8.generate_procedure;
    end;

make_interface.generate_interface;                           -- Step 12

end Dixie_Interface_View;
```

4.6 RESULTS OF OPERATION

The results of executing the Dixie_Interface_View procedure are the generated abstract interface specification and body, Dixie_Abs_Interface, the Ada specification of the SQL module, Dixie_Conc_Interface, and the text file Dixie_Dbms_Specific. Table 4.4 illustrates these generated products.

Table 4.4. Generated Products

Dixie_Abs_Interface - Specification

```
with dixie_employee_def_pkg;
use dixie_employee_def_pkg;
with dixie_salary_def_pkg;
use dixie_salary_def_pkg;
with sql_standard;
use sql_standard;
package dixie_abs_interface is

    type valid_status_result_type is
        (row_not_found, cursor_already_open, no_duplicates, cursor_not_open);

    type emp_rec is record
      emp_id : employee_id_not_null;
      emp_lastname : name_not_null;
      emp_firstname : name_not_null;
      emp_address : address_not_null;
      emp_weight : weight_not_null;
    end record;

    type sal_rec is record
     new_emp_id : employee_id_not_null;
     new_emp_sal : salary_not_null;
     new_emp_pts : points_type;
    end record;

    procedure startup;

    procedure keep_it;

    procedure close_emp_cursor;

    procedure trash_it;

    procedure open_emp_cursor(in_emp_id : in employee_id_not_null;
    result : out valid_status_result_type);
    procedure emp_fetch(fetch_rec : in out emp_rec;
                 fetch_done : out valid_status_result_type);

    procedure new_sal_row(insert_rec : in sal_rec;
      insert_done : out valid_status_result_type);
    procedure add_year(in_emp_id : in employee_id_not_null;
      change_done : out boolean);

end dixie_abs_interface;
```

Dixie_Abs_Interface - Body

```
with sql_communications_pkg, sql_database_error_pkg, conversions,
dixie_conc_interface;
use sql_communications_pkg, sql_database_error_pkg, conversions;

package body dixie_abs_interface is

  use points_ops;

  row_not_found_value : constant :=  100;
  cursor_already_open_value : constant :=  1001;
  no_duplicates_value : constant := -803;
  cursor_not_open_value : constant := -501;

  procedure startup is
    begin
      dixie_conc_interface.set_transaction (sqlcode);
      if sqlcode /= 0 then
        process_database_error;
        raise sql_database_error;
      end if;
  end startup;

  procedure keep_it is
  begin
      dixie_conc_interface.commit_transaction (sqlcode);
      if sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
    end if;
end keep_it;

procedure close_emp_cursor is
begin
    dixie_conc_interface.close_emp_cursor (sqlcode);
    if sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
    end if;
end close_emp_cursor;

procedure trash_it is
begin
    dixie_conc_interface.rollback_transaction (sqlcode);
    if sqlcode /= 0 then
      process_database_error;
        raise sql_database_error;
    end if;
end trash_it;

procedure open_emp_cursor(in_emp_id : in employee_id_not_null;
                          result : out valid_status_result_type) is
begin
    dixie_conc_interface.open_employees_cursor (char(in_emp_id), sqlcode);
    if sqlcode = cursor_already_open_value then
      result := cursor_already_open;
    elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
    end if;
end open_emp_cursor;
```

```
procedure emp_fetch(fetch_rec : in out emp_rec;
                    fetch_done : out valid_status_result_type) is
begin
    dixie_conc_interface.fetch_employees (char(fetch_rec.emp_id),
        char(fetch_rec.emp_lastname), char(fetch_rec.emp_firstname),
        char(fetch_rec.emp_address), real(fetch_rec.emp_weight), sqlcode);
    if sqlcode = row_not_found_value then
      fetch_done := row_not_found;
    elsif sqlcode = cursor_not_open_value then
      fetch_done := cursor_not_open;
    elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
    end if;
end emp_fetch;

procedure new_sal_row(insert_rec : in sal_rec;
                      insert_done : out valid_status_result_type) is
    new_emp_pts_c : int;
    new_emp_pts_indic : indicator_type;
begin
    if is_null(insert_rec.new_emp_pts)
      then new_emp_pts_indic := -1;
    else new_emp_pts_indic := 0;
      new_emp_pts_c := int(without_null_base(insert_rec.new_emp_pts));
    end if;
    dixie_conc_interface.insert_sal_hist_row (char(insert_rec.new_emp_id),
        real(insert_rec.new_emp_sal), new_emp_pts_c, new_emp_pts_indic,
        sqlcode);
    if sqlcode = no_duplicates_value then
      insert_done := no_duplicates;
    elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
    end if;
end new_sal_row;
```

```
  procedure add_year(in_emp_id : in employee_id_not_null;
                     change_done : out boolean) is
  begin
      dixie_conc_interface.update_job_hist ( char(in_emp_id), sqlcode);
      if sqlcode /= 0 then
        change_done := false;
      else change_done := true;
      end if;
  end add_year;

end dixie_abs_interface;
```

Dixie_Conc_Interface - Specification

```
with sql_standard; use sql_standard;

package dixie_conc_interface is

  procedure set_transaction(sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, set_transaction);

  procedure commit_transaction(sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, commit_transaction);

  procedure close_emp_cursor(sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, close_emp_cursor);

  procedure rollback_transaction(sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, rollback_transaction);

  procedure open_employees_cursor(in_emp_id: in char; sqlcode : out
            sql_standard.sqlcode_type);
  pragma interface (sql, open_employees_cursor);

  procedure fetch_employees( emp_id: in out char; emp_lastname: in out char;
            emp_firstname: in out char; emp_address: in out char;
            emp_weight: in out real;
            sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, fetch_employees);

  procedure insert_sal_hist_row( new_emp_id: in char; new_emp_sal: in real;
            new_emp_pts : in out int;
            new_emp_pts_indic : in sql_standard.indicator_type;
            sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, insert_sal_hist_row);

  procedure update_job_hist(in_emp_id: in char;
                            sqlcode : out sql_standard.sqlcode_type);
  pragma interface (sql, update_job_hist);

end dixie_conc_interface;
```

Dixie_Dbms_Specific - Text File

```
set_transaction sqlcode out sqlcode_type
commit_transaction sqlcode out sqlcode_type
close_emp_cursor sqlcode out sqlcode_type
rollback_transaction sqlcode out sqlcode_type
open_employees_cursor in_emp_id in char 5
open_employees_cursor sqlcode out sqlcode_type
fetch_employees emp_id in_out char 5
fetch_employees emp_lastname in_out char 10
fetch_employees emp_firstname in_out char 10
fetch_employees emp_address in_out char 30
fetch_employees emp_weight in_out real
fetch_employees sqlcode out sqlcode_type
insert_sal_hist_row new_emp_id in char 5
insert_sal_hist_row new_emp_sal in real
insert_sal_hist_row new_emp_pts in_out int
insert_sal_hist_row new_emp_pts_indic in indicator_type
insert_sal_hist_row sqlcode out sqlcode_type
update_job_hist in_emp_id in char 5
update_job_hist sqlcode out sqlcode_type
```

These packages are to be compiled with no further coding. If DBMS specific modifications are required for the interface due to DBMS implementation incompatibilities with the SAME standard, the text file containing information needed for these modifications and the interface packages are handed over to the person responsible for such modifications. Steps and guidelines for modifying the interface packages for use with the Digital RDB DBMS are provided in Section 6. If no modifications are required, then link the concrete interface specification to the SQL module. Steps for associating the concrete interface Ada specification to the SQL module are implementation dependent. Section 6 contains instructions for linking using the Digital ACS Ada language facility. After linking the specification to the module, make the interface packages available to the application programmer.

4.7 SUMMARY

In summary, the tasks shown in Table 4.5 must be performed during interface definition and generation.

Table 4.5. Task Summary

Task	Outcome
1. Compile both specification and body of `Concrete_Package` and `Abstract_Interface_Generator` against `Base_Specific_Domains`.	`Concrete_Package` and `Abstract_Interface_Generator` compiled with the valid domains.
2. Using the SQL module, determine the necessary row records, procedures and expected errors.	
3. Modify the SAME packages `Sql_Communications_Pkg` and `Sql_Database_Error_Pkg` for use with the DBMS and application (See DEC instructions in Section 6).	DBMS and application specific error processing and recovery capabilities.
4. Write an Ada procedure using `Abstract_Interface_Generator` to describe the interface. Use the template provided in Table 4.2.	
5. Compile and execute the procedure.	Four files: (1) the abstract interface specification, (2) the abstract interface body, (3) the concrete interface specification, and (4) a text file containing information needed to modify the generated interface for a particular implementation
6. Compile the interface packages.	Potentially semantically correct abstract interface and concrete interface specification to support the application.
7. If necessary, modify the interface for use with a specific DBMS using the DBMS specific file. (See DEC instructions in Section 6).	DBMS-compatible interface packages.
8. Link the concrete interface	Application specific SAME interface specification to the SQL module.(See DEC instructions in Section 6).

Section 5
Usage Instructions for the Application Programmer

5.1 GENERAL PREPARATION

At this stage, the application programmer has the compiled versions of the domain package specifications; the abstract interface specification and body, the concrete interface specification and the compiled, linked SQL module. The programmer is now ready to use these packages with the Ada application.

5.2 INPUTS FOR THE ADA APPLICATION PROGRAM

Given the Ada procedures in the abstract interface and their corresponding data manipulation statements in the SQL module, the application programmer now has access to the DBMS through the application. During application development, the programmer can use any of the procedures provided by the interface just as any other standard Ada procedure or function. However, all parameters to these procedures must be of a type declared in one of the domain packages or in the abstract interface, or of type boolean.

The following paragraphs provide the steps · ıd guidelines for writing an Ada application, `Ada_Application`, that uses a SAME interface.

5.3 STEPS AND GUIDELINES FOR THE ADA APPLICATION

To use a SAME interface, the application programmer always follows the same basic steps. These steps take the form of writing an Ada application that imports one or more domain packages containing data types and the newly generated abstract interface.

The following numbered steps are described in the order in which the application programmer writes the Ada code. Where appropriate, each step includes example code, and in some instances, specific code that implements the step. A generalized Ada application, also showing the steps, is shown in Table 5.1. Note that italics indicate user-supplied names. Names can be any valid Ada name but must be used consistently throughout the procedure.

Step 1. Import 2 or more packages:

`Domain_Definition_Pkg(s)` => One or more packages containing declarations of valid domains. Name and content of this package(s) is determined by the interface programmer. This package(s) must always be imported.

`Abstract_Interface(s)` => One or more packages containing the abstract interface portion of the SAME interface created for this application.
Name and content of this package(s) is determined by the interface programmer. This package(s) mı·ˢᵗ always be imported.

```
with domain_definition_pkg_1; use domain_definition_pkg_1;
with domain_definition_pkg_2; use domain_definition_pkg_2;
with domain_definition_pkg_3; use domain_definition_pkg_3;
-- Additional domain packages here...
with abstract_interface_1; use abstract_interface_1;
with abstract_interface_2; use abstract_interface_2;
-- Additional abstract interfaces here...

Example->
with parts_definition_pkg; use parts_definition_pkg;
with supplier_definition_pkg; use supplier_definition_pkg;
with product_abs_interface; use product_abs_interface;
```

Step 2. Create an Ada main procedure.

```
procedure Ada_Application is
  begin
end Ada_Application;
```

Step 3. Within this procedure declare all interface-required row records or string indicator records to be used by interface procedures taking row records as parameters. All valid row record types are declared in the abstract_interface specification.

```
declare
Row_Record_Name1 : Row_Record_Type1;
Row_Record_Name2 : Row_Record_Type2;...

Example->
Parts_Record : Parts_Record_Type;
Northern_Supplier_Record : Supplier_Record_Type;
```

Step 4. Declare all interface-required single parameters of domain type to be used by interface procedures taking single, domain-based parameters. All valid domain types are declared in the domain packages. If one of the SAME standard types' `assign` procedures will be used on a parameter of a null bearing domain type, you must declare another parameter of the corresponding not-null bearing domain type for conversion use (see Paragraph 3.4.1).

```
Domain_Parameter_Name1 : Domain_Type1;
Domain_Parameter_Name2 : Domain_Type2;...

Example->
declare
Part_Name : Product_Name_Not_Null;
Part_Price : Part_Price_Type;-- Null bearing numeric domain
Part_Price_nn : Part_Price_Not_Null;--Conversion parameter
```

Step 5. Declare all interface-required error condition result parameters. These result parameters will be used by the interface procedures to return expected error conditions to the application. Error condition result parameters are always of type `Valid_Status_Result_Type`.

```
Error_Name1 : Valid_Status_Result_Type;
```

```
Error_Name2 : Valid_Status_Result_Type;...

Example->
Result1 : Valid_Status_Result_Type;
Result2 : Valid_Status_Result_Type;
```

Step 6. Declare all interface-required parameters of boolean type to be used by interface procedures returning result information to the application.

```
Boolean_Result_Name1 : Boolean;
Boolean_Result_Name2 : Boolean;...

Example->
Part_Exists : Boolean;
Positive_Result : Boolean;
```

Step 7. Make all other application-required declarations and then begin coding the rest of the application, calling the interface procedures when needed as you would any other standard Ada procedures. Invalid interface procedure syntax and semantics will be caught during application compilation.

Step 8. Compile the application and correct any indicated errors.

Step 9. Execute the application.

All of the interface-required Ada code for this application is shown in a generalized form in Table 5.1. When writing an Ada application using a SAME interface, use this table as a guide substituting required names and values for any expressions in italics.

Table 5.1. Generalized Ada Application

```
with domain_definition_pkg_1; use domain_definition_pkg_1;          -- Step 1
with domain_definition_pkg_2; use domain_definition_pkg_2;
with domain_definition_pkg_3; use domain_definition_pkg_3;
-- Additional domain packages here...
with abstract_interface_1; use abstract_interface_1;
with abstract_interface_2; use abstract_interface_2;
-- Additional abstract interfaces here...

procedure Ada_Application is                                        -- Step 2

  declare
    Row_Record_Name1 : Row_Record_Type1;                            -- Step 3

    Row_Record_Name2 : Row_Record_Type2;...
    -- Additional record type declarations here...
    Domain_Parameter_Name1 : Domain_Type1;                        -- Step 4
    Domain_Parameter_Name2 : Domain_Type2;...
    -- Additional domain type declarations here...

    Error_Name1 : Valid_Status_Result_Type;                       -- Step 5
    Error_Name2 : Valid_Status_Result_Type;...
    -- Additional expected error type declarations here...

    Boolean_Result_Name1 : Boolean;                               -- Step 6
    Boolean_Result_Name2 : Boolean;...
    -- Additional boolean result type declarations here...

    -- Additional application-required declarations here...       -- Step 7

  begin
    -- Application statements and calls to interface procedures here...

end Ada_Application;
```

5.4 SAMPLE ADA APPLICATION

The SQL module shown in Table 3.1, the generated abstract domain packages shown in Table 3.4, and the generated abstract interface and concrete interface specification shown in Table 4.4 together provide the interface support for the Ada application shown in Table 5.2. The procedures in the abstract interface appear in bold-face type within the application body.

Table 5.2. Sample Ada Application Dixie_Application

```
--  This application imports the abstract inte  ace dixie_abs_interface
--  and the abstract domain packages to access the dixie_db data base.

with dixie_employee_def_pkg; use dixie_employee_def_pkg;        -- Step 1
with dixie_salary_def_pkg; use dixie_salary_def_pkg;
with dixie_abs_interface; use dixie_abs_interface;
with text_io; use text_io;
with string_pack; use string_pack;

procedure dixie_application is                                 -- Step 2

  personnel_record : emp_rec;                                  -- Step 3
  business_record : sal_rec;
```

```
id_number : employee_id_not_null;                              -- Step 4
-- merit_pts_nn is used for type conversion between the null bearing
-- points_type and the non null bearing points_not_null so that the
-- points_ops.assign procedure can be used on a variable of this type
merit_pts_nn : points_not_null;

error_result : valid_status_result_type;                       -- Step 5

boolean_result : boolean;                                      -- Step 6

-- Variables and instantiations for general program use        -- Step 7
valid_code : boolean := false;
option_entry : character;
exit_code : constant character := '8';
see_employ_code : constant character := '1';
add_job_code : constant character := '2';
new_sal_code : constant character := '3';
commit_code : constant character := '4';
rollback_code : constant character := '5';
release_screen : character;
weight_image, string_holder : string(1..100) := (others => ' ');
last : natural;
trash : string (1..1) := (others => ' ');
package integer_io is new text_io.integer_io(integer);
package real_io is new float_io(float); use real_io;

-- This function gets the option code (either access dixie_db or exit
-- the program) from the user.
function option_input return character is
  begin
    valid_code := false;
    while not valid_code loop
      text_io.put_line(" ");
      put_line("Please enter an option number. Either: ");
      put_line("1 : to view a record from the employees table or");
      put_line
         ("2 : to add 1 year to years employed in the job_hist table or");
      put_line("3 : to insert a new record into the salary_hist table");
      put_line("4 : to commit the transaction");
      put_line("5 : to rollback the transaction");
      put_line("8 : to exit this program.");
      put_line(" ");
      put("Enter option here => ");
      get (option_entry);
      text_io.get_line(trash, last);
      case option_entry is
        when see_employ_code => valid_code := true;
        when add_job_code => valid_code := true;
        when new_sal_code => valid_code := true;
        when commit_code => valid_code := true;
        when rollback_code => valid_code := true;
        when exit_code => valid_code := true;
        when others =>
          put("Invalid option.  Type 'C' and press <RET> to continue.");
          get(release_screen);
      end case;
    end loop;
    return option_entry;
  end option_input;

begin
    -- The main program will loop until the user requests option 8 or exit.
    put_line("**********Welcome to the Dixie_Db update program**********");
    put_line(" ");
    startup;
    loop
    if option_input = exit_code then
      exit;
    elsif option_entry = see_employ_code then
      put_line("Please enter an employee id number => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(id_number));
      open_emp_cursor(id_number,error_result);
      if error_result = cursor_already_open
      then close_emp_cursor;
           text_io.put_line("Try again");
      else emp_fetch(personnel_record, error_result);
           if error_result = row_not_found
          then text_io.put_line("No employee with this number found");
          else real_io.put (weight_image, float(personnel_record.emp_weight),
                           2,0);
              text_io.put_line(strip(string(personnel_record.emp_id)) & " " &
              strip(string(personnel_record.emp_lastname)) & " " &
              strip(string(personnel_record.emp_firstname)) & " " &
              strip(string(personnel_record.emp_address)) & " " &
              strip(weight_image));
          end if;
      end if;
      close_emp_cursor;
      put_line(" ");

    elsif option_entry = add_job_code then
      put_line("Please enter an employee id number => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(id_number));
      add_year(id_number, boolean_result);
      put_line(" ");
      if boolean_result = false
      then text_io.put_line("Year not updated");
          put_line(" ");
      end if;

    elsif option_entry = new_sal_code then
      put_line("Enter an employee id number (5 characters) .. not null => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(business_record.new_emp_id));
      put_line ("Enter earnings for employee (0.00 to 99000.00) .. not null =>");
      real_io.get(float(business_record.new_emp_sal));
      text_io.get_line(trash, last);
      put_line
        ("Enter merit points for employee (0 to 99) .. return for null => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder,last);
      if last = 0  -- they are inputting a null
      then points_ops.assign(business_record.new_emp_pts, null_sql_int);
      else
```

```
        integer_io.get(string_holder, integer(merit_pts_nn), last);
        points_ops.assign(business_record.new_emp_pts,
                          points_ops.with_null(merit_pts_nn));
      end if;
      new_sal_row(business_record, error_result);
      put_line(" ");

    elsif option_entry = commit_code then
      keep_it;

    elsif option_entry = rollback_code then
      trash_it;

    end if;
    end loop;
end dixie_application;
```

5.5 RESULTS OF OPERATION

The Ada application using the generated domain packages and the abstract interface has access to an SQL module and, through that module, access to a DBMS. Because creation of the domain packages and the interface packages is implemented through strong typing definitions and successive generic instantiations to match the SQL module, syntactic and semantic consistency is ensured between those packages and the SQL module. Because the SQL module is a compiled module, all SQL data manipulation statements are ensured to be both syntactically and semantically correct. Therefore, the application program is ensured syntactically and semantically correct access to the DBMS.

5.6 SUMMARY

In summary, the steps shown in Table 5.3 are taken to use the interface in an Ada application.

Table 5.3. Steps Summary

Steps	Outcome
1. Receive compiled versions of the abstract domain package(s) and the abstract interface(s).	
2. Write an Ada application importing the abstract domain package(s) and the abstract interface(s).	Ability to invoke Ada procedures that access a DBMS.
3. Compile the application	Syntactic or semantic errors of interface procedures caught during compilation
4. Execute the application.	

Section 6
Operating Instructions

6.1 OVERVIEW

This section contains operational information for using the Ada/SQL binding to interface with the Digital RDB DBMS. This section is organized into four parts. After this overview, paragraph 6.2 provides installation requirements and instructions. Paragraph 6.3 contains guidelines for using RDB/VMS error processing capabilities in the abstract interface. Paragraph 6.4 discusses adapting the `rdb_specific` procedure for use with other DBMS systems.

6.2 INSTALLATION

The following subparagraphs outline the hardware and software requirements for installation and provide step-by-step procedures for installation.

6.2.1 Hardware Requirements

The hardware requirements for this implementation of the Ada/SQL binding consist of the following:

- A Digital VAX with a minimum of 8Mb physical memory
- Sufficient disk space for
  –The RDB software
  –The DEC Ada compilation software
  –The Ada/SQL binding program software
  –Swap space for Ada compilations
- Capability for loading software—cartridge tape device

6.2.2 Software Requirements

The exact locations for the following software are not important. However, in the following paragraphs it will be assumed that the user has visibility into RDB DBMS, DEC Ada compiler, and VMS executable commands. It is also assumed that an RDB data base has been created, loaded, and placed within a working directory together with the Ada/SQL binding.

- Ada/SQL, Version 1.0 - Ada/SQL binding tool
- DEC VMS, Version 5.2 - operating system
- DEC Ada,Version 1.4 - Ada compiler
- DEC RDB, Version 3.0 - DMBS

See the DEC reference manuals *Vax Rdb/VMS Reference Manual* [3] or *Developing Ada Programs on Vax/VMS* [4] for further information concerning installation and use of DEC Ada or DEC RDB.

6.2.3 Installation of Ada/SQL Software

Follow these steps to install the Ada/SQL binding tool :

Step 1. Copy all of the directories and objects on the delivered tape into an equivalent directory structure. The top-level delivered directory is `[project.c2010]`. This directory should contain the following:

- `ada_units.dir`—subdirectory containing all components of the Ada/SQL binding tool, executable procedures for testing the binding and the file, `ada_units.ord` showing unit compilation order. Ada package specifications have the file extension `.ads` and Ada package bodies or procedures have the file extension `.adb`. A description of each unit in this directory is provided in Appendix A.
- `dixie_db.rdb`—a fully created data base supporting the test procedures in `ada_units.dir`. The file `dixie_db.snp` is simply a snapshot of the data base.
- `dixie_conc_interface.sqlmod`—a complete SQL module supporting the test procedures in `ada_units.dir`.
- `domain_template.txt`—a program template with instructions for creating and defining the domain packages
- `interface_template.txt`—a program template with instructions for creating and defining the interface packages
- `readme.nte`—a text file containing additional information concerning the file`[project.c2010.ada_units]ada_units.ord`.
- `test_instr.tst`—instructions for using the test procedures contained in `[project.c2010.ada_units]`.
- `vdd.txt`—the Ada/SQL binding version description document

Step 2. Create a DEC Ada ACS library in directory `[project.c2010.ada_units]` with the DEC Ada command:

```
{acs create lib [project.c2010.ada_units.adalib]}
```

Step 3. Set the default Ada library to this library with the command:

```
{acs set library [project.c2010.ada_units.adalib]}
```

Step 4. Following the compilation order given in `ada_units.ord` (also in Appendix A) compile all units in directory `[project.c2010.ada_units]` up to and including unit `abstract_domain_generator.ada` using the DEC ada compile command `{ada}`.

If you are installing on a system with another Ada compiler, you must follow instructions for compiling and executing Ada procedures and packages for that system.

6.2 USE OF THE BINDING WITH DEC ADA AND RDB

The directory `[project.c2010.ada_units]` contains three executable procedures that serve as both tests and examples for using the binding. These procedures are, in fact, the same procedures shown in Sections 3 through 5 as examples.

- `dixie_domain_view.ada`—test domain definition procedure.
- `dixie_interface_view.ada`—test interface definition procedure.
- `dixie_application.ada`—test application.

The following steps, using these test procedures as examples, are taken when using the Ada/SQL binding with the RDM DBMS.

Step 1. Go into the directory `[project.c2010.ada_units]`. Copy both the data base and the SQL module to be used by the Ada application into this directory. For this example, copy `dixie_db.rdb`, `dixie_db.snp` and `dixie_conc_interface.sqlmod` into this directory.

Step 2. Compile, link, and execute the domain definition procedure to generate the abstract domain package specification(s) and the specification `base_specific_domains`. Use procedure `dixie_domain_view.ada` for this example.

```
{ada dixie_domain_view.ada}
{acs link dixie_domain_view}
{run dixie_domain_view}
```

Step 3. Compile the generated products. For this example, compile the domain packages `dixie_employee_def_pkg_.ada` and `dixie_salary_def_pkg_.ada` and `base_specific_domains_.ada`.

```
{ada dixie_employee_def_pkg_.ada}
{ada dixie_salary_def_pkg_.ada}
{ada base_specific_domains_.ada}
```

Step 4. Compile the `abstract_interface_generator` and `concrete_package` packages against this specific base_specific_domains.

```
{ada concrete_package_.ada}
{ada concrete_package.ada}
{ada abstract_interface_generator_.ada
{ada abstract_interface_generator.ada}
```

Step 5. Compile, link, and execute the interface definition procedure to generate the abstract interface specification and body, the concrete interface specification and a text file containing interface information. Use procedure `dixie_interface_view.ada` for this example.

```
{ada dixie_interface_view.ada}
{acs link dixie_interface_view}
{run dixie_interface_view}
```

Step 6. Compile all generated products except the text file having the file name extension `.txt`. For this example, compile the abstract interface specification, `dixie_abs_interface_.ada`, the concrete interface specification `dixie_conc_interface_.ada` and the body, `dixie_abs_interface.ada`. Do not compile `dixie_dbms_specifc.txt`.

```
{ada dixie_abs_interface_.ada}
{ada dixie_conc_interface_.ada}
{ada dixie_abs_interface.ada}
```

Step 7. Compile, link and execute `rdb_specifc` to generate three files. Two of the files are the Ada specification and body that will replace the original concrete interface specification for the abstract interface. This procedure will assign the original concrete specification's name to both this new specification and body (so that no units dependent on the original specification require modification). The other file is an Ada specification that will replace the original concrete specification for the SQL module. It is this new specification that will contain the actual RDB compatible interface procedures.

```
{ada rdb_specifc.ada}
```

When executing `rdb_specific`, the user will be asked to provide the filename of the original concrete interface specification, the filename of the text file containing interface information, and a name to be given to the new RDB-compatible concrete interface specification. An execution error will occur if either the original specification or the text file are incorrectly named or not present in the directory. At the end of the program, the user will be asked if additional specifications require replacement. A positive response will cause the program to loop through again. For this example, respond to the program prompts as follows:

```
Input name of the file containing original concrete_spec =>
{dixie_conc_interface_.ada <return>}

Input name of the file containing dbms specific information =>
{dixie_dbms_specifc.txt <return>}

Input the name of the actual concrete_spec =>
{dixie_rdb_interface_.ada <return>}

Any more abstract specifications?  Answer y or n =>
{n <return>}
```

Step 8. Compile all generated products. For this example, compile the new RDB-compatible concrete interface specification `dixie_rdb_interface_.ada`, the new concrete interface specification, `dixie_conc_interface_.ada`, and the body, `dixie_conc_ interface.ada`. The abstract_interface specification and body will need to be recompiled to reflect the newly generated concrete_interface.

```
{ada dixie_rdb_interface_.ada}
{ada dixie_conc_interface_.ada}
{ada dixie_conc_interface.ada}
{ada.dixie_abs_interface_.ada}
{ada.dixie_abs_interface.ada}
```

Step 9. Compile the SQL module, `dixie_conc_interface.sqlmod` for this example, using the RDB SQL module language compiler with following command:

```
{run sys$system:sql$mod <return>}
Input file=>
{dixie_conc_interface.sqlmod}
```

After compilation the SQL module object file will appear in the directory, for example `dixie_conc_interface.obj`. This is the file that must be linked to the concrete interface Ada specification.

Step 10. Define the link library as:

```
{define lnk$library sys$library:sql$user.olb}
```

Step 11. Link the SQL module to the Ada specification using the DEC Ada ACS command `copy foreign`. More than one concrete interface specification can be linked a given SQL module. To link `dixie_conc_interface.obj` to `dixie_rdb_interface_.ada`, enter the command:

```
{acs copy foreign dixie_conc_interface.obj
     dixie_rdb_interface}
```

Note: dixie_rdb_interface is a unit name with no file extension.

Step 12. Compile, link and execute the application. Use the procedure `dixie_application.ada` in this example:

```
{ada dixie_application.ada}
{acs link dixie_application}
{run dixie_application}
```

6.3 ERROR PROCESSING WITH RDB

During interface development, the interface programmer determines what types of errors are expected from the data base during application program execution. These expected errors are handled within the application program. In the case of unexpected errors, however, SAME provides a place for error processing procedures within two packages:

- `Sql_Communications_Pkg`
- `Sql_Database_Error_Pkg`.

These packages are intended to be modified by the interface programmer to meet the error processing requirements of the application and the platform.

`Sql_Communications_Pkg` contains a function, `sql_database_error_message` for DBMS error message retrieval and an Ada exception, `sql_database_error`. `Sql_Database_Error_Pkg` contains a procedure, `process_database_error`, for error processing and recovery. If an unexpected error occurs during application execution, the abstract interface body first calls the procedure `process_database_error` and then raises the `sql_database_error` exception. The procedure `process_database_error` should minimally make a call to the function `sql_database_error_message` to print out what unexpected error occurred. Other processing might include a call to the rollback or commit procedures. Once the error `sql_database_error` has be raised, it propagates to the user thereby alerting the application that an unexpected error has occurred. Thus, all unexpected error processing and recovery procedures or functions should be contained within the `process_database_error` procedure.

The Vax RDB/VMS system provides a large number of useful error processing procedures. Refer to the *Vax Rdb/VMS Reference Manual* [3] for complete information concerning Vax RDB/VMS error processing capabilities. For the purpose of illustration, the following examples show modified versions of the `Sql_Communications_Pkg` body (Table 6.1) and the `Sql_Database_Error_Pkg` body (Table 6.2). These modified package bodies use the RDB/VMS service procedure SYS$GET_ERROR_TEXT to retrieve and output a descriptive error message from the RDB DBMS.

Table 6.1. Modified Package Body `Sql_Communications_Pkg`

```
-- This version of sql_communications_pkg.ada is modified for use with the
-- DEC RDB DBMS.  The function sql_database_error_message
-- will get an RDB error message using the RDB/VMS supplied procedure
-- SQL$GET_ERROR_TEXT.

with sql_system; use sql_system;
with text_io; use text_io;
with condition_handling; -- RDB/VMS package providing error condition system
                         -- service functions.
with system;             -- RDB/VMS package providing general system service
                         -- functions.
package body sql_communications_pkg is

-- The following variables are part of the standard sql_communications_pkg
-- They are used here in slightly modified form as parameters in the
-- SQL$GET_ERROR_TEXT procedure.

  function sql_database_error_message return sql_char_not_null is
    message_buffer : string(1 .. 256);  -- Always a string this length
    len : short_integer;                -- Always a short_integer

    -- Declaration of SQL$GET_ERROR_TEXT.
    procedure sql_get_error_text(error_text : out string;
    error_text_len : out short_integer);
    pragma interface(SQL, sql_get_error_text);
    pragma import_procedure(internal => sql_get_error_text,
                            external => "SQL$GET_ERROR_TEXT",
                            parameter_types => (string, short_integer),
                            mechanism => (descriptor, reference));
  begin
    -- This function gets the RDB DBMS error message and passes it on to
    -- the sql_database_error_pkg for output.
    sql_get_error_text(message_buffer, len);
    return (sql_char_not_null(message_buffer));

  end sql_database_error_message;

end sql_communications_pkg;
```

Table 6.2. Modified Package Body Sql_Database_Error_Pkg

```
- This version of sql_database_error_pkg.ada is modified for use with the
-- DEC RDB DBMS.  RDB/VMS provides various functions and procedures for
-- processing data base errors. See the Vax RDB/VMS reference manual for
-- information concerning error processing capabilities.

with text_io, sql_communications_pkg, sql_char_pkg;
use text_io, sql_communications_pkg, sql_char_pkg;

package body sql_database_error_pkg is

  procedure process_database_error is

    begin
      -- This procedure is always called in the abstract interface in response
      -- to some unexpected database exception (as opposed to one of the
      -- expected errors declared in the abstract interface).

      -- This procedure may be modified per the needs of the abstract
      -- interface developer.  All needed error processing procedures and
      -- functions, whether supplied by RDB/VMS or developed by the interface
      -- programmer should be placed within the body of this procedure.
      -- This is a minimal implementation of an error processing procedure.
      -- This procedure gets a descriptive error message from the DBMS
      -- (through sql_communications_pkg) and displays it on standard output.

      put_line (to_string (sql_char_not_null(sql_database_error_message)));

  end process_database_error;
end sql_database_error_pkg;
```

6.4 rdb_specific PROCEDURE ADAPTIONS

Using the SAME method with any specific DBMS implementation very probably will require some modification to the concrete interface. In the case of a DBMS with no SQL module language compiler, for example, the concrete interface will have to simulate a compiled SQL module and provide DBMS supplied programmatic access instead, such as a library of C routines. SAME standard data types may also be incompatible with a particular DBMS (as with the Digital RDB DBMS) requiring conversion of the SAME standard data types declared in the concrete interface to DBMS required data types. Although each DBMS implementation will require its own particular concrete interface modification, the `rdb_specific` procedure, for use with the Digital RDB DBMS, can be adapted to work with different DBMS implementations.

The `rdb_specific` procedure uses the information contained in the original concrete interface Ada specification and the the *`Dbms_Specific`* text file. Both of these files are automatically generated along with the abstract interface. The concrete interface specification contains the specifications of the Ada procedures representing SQL procedures and the pragma interface statement for each procedure. *`Dbms_Specific`* contains procedure descriptions: procedure name, parameter name, parameter mode, parameter type, and with `char` type parameters, their length. By using these two files, the `rdb_specific` procedure has enough information to convert procedures and data types to be compatible with the RDB DBMS. In the process, three new files are generated to replace the original concrete interface specification. Any adaptation of the `rdb_specific` will also use this information for procedure and data type conversions and will generally generate these three files.

The first two files are an Ada package, specification and body, that will substitute for the original concrete interface specification. This package always bears the same name as the original concrete specification so that no modification to the calling abstract interface is required. The third file is an Ada specification that contains the actual Ada procedures and data types that are compatible with the RDB DBMS and is the specification that represents the SQL module.

In summary, the `rdb_specific` procedure takes information from the original concrete interface specification and *`Dbms_Specific`* text file, converts the procedures and data types, and outputs three replacement files. Therefore, adapting the `rdb_specific` procedure for use with another DBMS implementation basically requires changing the `rdb_specific`'s file output statements.

For instance, if a DBMS supplies data base access through a library of C routines rather than an SQL module language compiler, the `rdb_specific` procedure could be changed to generate the following three replacement files. The first file will be a package specification containing Ada procedures that represent the needed C routines and associated pragma interface statements to C. These C routines will in turn, provide access to the required SQL statements, and like the SQL module, will be developed separately. The second and third files will comprise the package specification and body to be imported by the abstract interface instead of the original concrete specification. This package specification will declare the Ada procedures that represent the SQL statements from the abstract interface side, and it will look like the original concrete specification with the pragma interfaces removed. In the package body each procedure will call the matching Ada procedure specification and pragma interface to C in the first specification.

APPENDIX A
Component Description and Compilation Order

A.1 FUNCTIONAL AREAS

This appendix describes the components of the Ada/SQL binding in terms of functionality and, where appropriate, input and output. Names in italics indicate user-supplied names. The components are grouped into four overall functional areas. The first group of components provides domain definition and generation capabilities. The second group provides abstract interface definition and generation capabilities. The third group is comprised of general purpose components that support this binding by providing additional functionality to the generators and global declarations. The fourth group consists of all components that are part of the standard SAME method. The final portion of Appendix A provides the compilation order of the components.

A.2 DOMAIN DEFINITION AND GENERATION

`Abstract_Domain_Generator` is a generic Ada package that contains several layers of nested generics that are to be instantiated by an Ada procedure, the *`Domain_View`*, written by the interface programmer. Based on the information contained in the *`Domain_View`*, the `Abstract_Domain_Generator` will generate a set of Ada specifications, the domain packages, containing declarations of all domains needed by the application programmer.

Inputs to `Abstract_Domain_Generator`:

The *`Domain_View`* is an Ada procedure written by the interface programmer following the guidelines in Section 3. When this procedure is compiled and executed, one or more Ada specifications representing the domains are automatically generated. This procedure is written specifically for each application or set of applications.

Outputs from `Abstract_Domain_Generator`:

- The Domain packages generated by the instantiation of the *`Domain_View`*. These packages are syntactically and semantically correct Ada and adhere to the SAME methodology. No further coding or modification is required. These packages will be `withed` by both the application and the abstract interface and give both packages access to data types represented by the domains.
- `Base_Specific_Domains` is another package created by the `Abstract_Domain_Generator` during domain package generation. This Ada specification contains information about the domain packages and the individual domains that will be used by `Abstract_ Interface_Generator`. This specification also requires no further coding or modification.

A.3 INTERFACE DEFINITION AND GENERATION

`Abstract_Interface_Generator` is a generic Ada package that contains several layers of nested generics that are to be instantiated by an Ada procedure, the *`Interface_View`*, written by the interface programmer. Based on the information contained in the *`Interface_View`*, `Abstract_Interface_Generator` will generate an Ada specification and body for a specific abstract interface and an Ada specification representing a specific and corresponding concrete SQL module.

Inputs to `Abstract_Interface_Generator`:

`Interface_View` is the second Ada procedure that must be written by the interface programmer following the directions given in Section 4. When this procedure is compiled and executed, an Ada specification and body of an abstract interface and an Ada specification of an SQL module are automatically generated. This procedure is specific to the services requested of the data base and particular rows in the data base and is written specifically for each application or set of applications.

Outputs from `Abstract_Interface_Generator`:

- *`Abstract_Interface`* is an Ada package, specification and body, automatically generated by the instantiation of the Abstract_Interface generic. This package will give an Ada application access to Ada procedures representing SQL data manipulation statements. This package is syntactically and semantically correct Ada and will adhere to the SAME methodology. No further coding or modification is required. This package will be withed by the application.

- *`Concrete_Interface`* is an Ada specification representing the SQL module that contains the SQL statements to implement services requested from the data base. This specification is complete, requiring no further coding or modifications.

- *`DBMS_Specific`* is an additional text file that contains all interface information, such as parameter length or type, necessary to successfully interface with various types of data base implementations. This text file is complete and requires no modifications.

A.4 GENERAL BINDING SUPPORT COMPONENTS

- `rdb_specific` is a special purpose Ada procedure to be used only with Digital RDB DBMS. Because some SAME required data types and RDB data types are incompatible, the SAME data types declared in the *`Concrete_Interface`* must be converted to their corresponding RDB data types. The `rdb_specific` procedure takes the original *`Concrete_Interface`* and the interface information contained in *`Dbms_Specific`* and generates three files. These files, in turn, replace the original concrete specification for use with the RDB data base. This procedure can be modified to work with different DBMS implementations.

  Inputs to `rdb_specifc`:

  - *`Dbms_Specific`* contains all interface information, such as parameter length or type, necessary to successfully interface with various types of data base implementations.

- *Concrete_Interface* is the Ada specification representing the SQL module that contains the SQL statements to implement services requested from the data base.

  Outputs from rdb_specifc:

  - *Concrete_Interface* is an Ada package, specification and body, automatically generated by executing the rdb_specific procedure. This entire package will substitute for the original *Concrete_Interface* specification. This package will always bear the same name as the original concrete specification. No further coding or modification is required. This package will be withed by both the application and the abstract interface instead of the original *Concrete_Interface*.

  - *Dbms_Required_Interface* is an Ada specification automatically generated by executing the rdb_specific procedure. This specification contains the actual Ada procedure declarations that are compatable with the RDB DBMS and is the specification that will represent the SQL module. *Dbms_Concrete_Interface* imports this package to obtain access to the SQL module. No further coding or modification is required.

- Concrete_Package is an Ada package that given an abstract domain, provides base type information. Like package Abstract_Interface_Generator, this package imports the specification Base_Specific_Domains.

- Conversions is an Ada package that tracks and supports conversions of null values returned in indicator variables.

- Domain_Template is a text file containing a code template and instructions for writting the procedure that defines and generates the domain packages.

- Enumeration_Image is an Ada generic function that takes an enumeration type and returns a string equivalent of an Ada enumeration declaration statement.

- Generator_Support contains the global enumeration types and constants used by the generators and the application program. Some values, such as maximum string lengths, can be modified to suit user preference or platform constraints.

- Interface_Template is a text file containing a code template and instructions for writting the procedure that defines and generates the abstract interface.

- Longest_Enum is an Ada generic function that takes an enumeration list and returns an integer representing the length of the longest enumerated value

- Linked_List is an Ada generic package that implements lists. List building, updating, and deleting, as well as logical list operations, are provided.

- String_Pack is an Ada package containing utility procedures and functions that operate on fixed and variable length strings.

- Variable_Io is an Ada generic package that takes a value representing the desired maximum length of a file output line and then controls outputs lines to meet this specification.

A.5 SAME STANDARD COMPONENTS

- `Sql_Standard` is an Ada specification containing Ada type definitions corresponding to SQL types with each type definition directly defining the SQL type of the same name. The types in `Sql_Standard` define the representation of SQL data to the Ada compiler, and all abstract domains and interface parameters are based on these types. The type definitions are DBMS implementation dependent.

- `Sql_System` is a supporting Ada specification that defines maximum string output length. This specification is DBMS implementation dependent.

- `Sql_Communications_Pkg` and `Sql_Database_Error_Pkg` are Ada packages that provide the interface programmer with error recovery processing. Both packages are to be modified to the requirements of the DBMS and the user. Note that `Sql_Communications _Pkg` is DBMS implementation dependent and `Sql_ Database_Error_Pkg` is application dependent.

- The following are Ada packages supporting logical operations of the types defined in `Sql_Standard`. These packages are implementation independent and should never be modified.

```
Sql_Boolean_Pkg
Sql_Char_Pkg
Sql_Double_Precision_Pkg
Sql_Enumeration_Pkg
Sql_Exceptions
Sql_Int_Pkg
Sql_Real_Pkg
Sql_Smallint_Pkg
To_Sql_Char_Not_Null
To_String
```

A.6 DOMAIN DEFINITION AND GENERATION

The following list of filenames represents the required compilation order for all components of the Ada/SQL binding. Filename extensions ending in `.ads` are Ada package specifications and those ending in `.adb` are Ada package bodies or Ada procedures. File names in italics such as *`domain_view.adb`* represent either files created by the interface program or files automatically generated by the execution of a procedure. The names of such files are, therefore, user-supplied.

```
enumeration_image.ads
string_pack.ads
string_pack.adb
enumeration_image.adb
generator_support.ads
longest_enum.ads
longest_enum.adb
linked_sets.ads
linked_sets.adb
sql_boolean_pkg.ads
sql_exceptions.ads
sql_boolean_pkg.adb
```

sql_system.ads	
sql_database_error_pkg.ads	
sql_standard.ads	
sql_char_pkg.ads	
sql_char_pkg.adb	
sql_communications_pkg.ads	
sql_communications_pkg.adb	
sql_database_error_pkg.ads	
variable_io.ads	
variable_io.adb	
sql_int_pkg.ads	
sql_int_pkg.adb	
sql_real_pkg.ads	
sql_real_pkg.adb	
sql_smallint_pkg.ads	
sql_smallint_pkg.adb	
to_string.adb	
to_sql_char_not_null.adb	
sql_double_precision_pkg.ads	
sql_double_precision_pkg.adb	
sql_enumeration_pkg.ads	
sql_enumeration_pkg.adb	
conversions.ads	
conversions.adb	
abstract_domain_generator.ads	
abstract_domain_generator.adb	
domain_view.adb	Executable procedure written by the interface programmer.
base_specific_domains.ads	Generated by executing domain_view.adb.
abstract_domain_package(s).ads	One or more packages generated by executing domain_view.adb.
concrete_package.ads	
concrete_package.adb	
abstract_interface_generator.ads	
abstract_interface_generator.adb	
interface_view.adb	Executable procedure written by the interface programmer.
abstract_interface.ads	Generated by executing interface_view.adb.
concrete_interface.ads	Generated by executing interface_view.adb.
abstract_interface.adb	Generated by executing interface_view.adb.
SQL MODULE	Compiled and linked to concrete_interface.adb.
user_application.adb	Executable procedure written by the application programmer.

If the generated interface requires DBMS-specific modifications, compile as before up to and including *abstract_interface*.adb and then use the compilation order below. Also, see Section 6.

rdb_specifc.adb	Executable procedure for modifying the interface
dbms_required_interface.ads	Generated by executing rdb_specifc.adb
dbms_concrete_interface.ads	Generated by executing rdb_specifc.adb
dbms_concrete_interface.adb	Generated by executing rdb_specifc.adb
abstract_interface.ads	Generated by executing interface_view.adb.
abstract_interface.adb	Generated by executing interface_view.adb.
SQL MODULE	Compiled and linked to dbms_required_interface.adb.
user_application.adb	Executable procedure written by the application programmer.

APPENDIX B
SAME SUPPORT EXCLUSIONS

The following list shows all exclusions to SAME support for this prototype implementation:

- **Decimal Types**—ANSI SQL supports the type decimal. The Ada programming language, however supports no directly analogous type. Furthermore, ANSI standard SQL, as described in [2], does not support decimal data in Ada programs. The SAME standard, therefore, does not provide a standard support package for null and non-null bearing decimal types and, therefore, this implementation also does not support Decimal Types.
- **Arbitrary Data Types**—SAME provides standard support for types in ANSI standard SQL. Many data base management systems extend the standard to other types. SAME documentation outlines the way a SAME user can extend the data typing facilities, however, this implementation does not support data typing extensions.
- **Enumeration Type Representation**—Whether enumeration values are represented as string literals or integer values is the responsibility of the abstract module. This implementation only provides string representation.

APPENDIX C EXECUTION ERROR MESSAGES AND ERROR CORRECTION

C.1 DOMAIN DEFINITION AND GENERATION EXECUTION ERRORS

C.1.1 Domain Package Errors

ERROR TYPE:	duplicate package name
message	`Error... Domain package` *`domain_package_name`* `has been defined more than once.`
correction	Multiple packages sharing the same name are not allowed. Declare a unique name for each domain package.
ERROR TYPE:	uncreated package
message	`Error... One or more domain packages has been declared but never described.`
correction	Domain packages that are declared but then not later described are not allowed. For every domain package name declared, completely define the content(domains) of that package.

C.1.2 Domain Errors

ERROR TYPE:	duplicate domain name
message	`Error...` *`domain_name`* `in domain package` *`domain_package_name`* `has been defined more than once.`
correction	Multiple domains sharing the same name are not allowed. Declare a unique name for each domain.
ERROR TYPE:	undefined domain
message	`Error... One or more domains in domain package` *`domain_package_name`* `has been declared but never described.`

correction	Domains that are declared but then not later described are not allowed. For every domain name declared for a domain package, completely define that domain.

ERROR TYPE:	illegal char subtype
message	`Error... Subtype based on domain of type char in domain package` *`domain_package_name`* `not allowed.`
correction	SAME does not allow char subtypes. Remove the char subtype declaration and definition.

ERROR TYPE:	undeclared parent domain
message	`Error... parent type for` *`domain_subtype_name`* `has not been previously defined. ... cannot declare a subtype unless parent type has been previously defined.`
correction	The SAME implementation requires that all derived subtypes be declared after their parent type. Declare the parent domain before declaring the derived subtype domain.

ERROR TYPE:	undefined parent domain
message	`Error...` *`parent_domain_name`* `has not been previously defined. ... cannot derive a new type unless parent domain has been has been previously defined`
correction	The SAME implementation requires that all derived subtypes be described after their parent type is described. Define the parent domain before defining the derived subtype domain.

C.2 INTERFACE DEFINITION AND GENERATION EXECUTION ERRORS

C.2.1 Error Definition Errors

ERROR TYPE:	unequal errors and SQLCODEs
message	`Error... Number of expected SQL errors does not match corresponding sql_code values given.`
correction	For every expected error declared, one and only one corresponding SQLCODE value must be given. Add or remove the incorrect SQLCODES.

ERROR TYPE:	too many errors for boolean result
message	`Error... A result parameter of type boolean has too many possible errors associated with it.`
correction	Boolean type results are with either no errors or at most, one (and only one) error. Remove all extra errors from association with this result parameter.

C.2.2 Record Definition Errors

ERROR TYPE:	duplicate record
message	`Error...` *record_name* `has been defined more than once.`
correction	Multiple records sharing the same name are not allowed. Declare a unique name for each record.

ERROR TYPE:	duplicate record component
message	`Error...` *record_component_name* `has been defined more than once.`
correction	Multiple record components sharing the same name are not allowed. Declare a unique name for each record component.

ERROR TYPE:	undefined record component
message	`Error...` *record_component_name* `in record` *record_name* `has been declared but never described.`
correction	Components that are declared but then not later described are not allowed. For every component name declared for a record, completely define that component.

C.2.3 Procedure Definition Errors

ERROR TYPE:	duplicate procedure
message	`Error...` *procedure_name* `has been defined more than once.`
correction	Multiple procedures sharing the same name are not allowed. Declare a unique name for each procedure.

ERROR TYPE:	duplicate procedure parameter
message	`Error...` *`procedure_parameter_name`* `of procedure` *`procedure_name`* `has been defined more than once.`
correction	Multiple procedure parameters sharing the same name are not allowed. Declare a unique name for each procedure parameter.
ERROR TYPE:	undefined procedure parameter
message	`Error...` *`procedure_parameter_name`* `in procedure` *`procedure_name`* `has been declared but never described.`
correction	Parameters that are declared but then not later described are not allowed. For every parameter name declared for a procedure, completely define that parameter.
ERROR TYPE:	multiple result parameters
message	`Error... Cannot provide more than one result parameter for procedure` *`procedure_name.`*
correction	Only one result parameter is allowed per procedure. Remove all but one result parameter from the procedure definition.
ERROR TYPE:	no result paramter
message	`Error... Must provide result parameter for procedure` *`procedure_name.`*
correction	This kind of procedure expects a result parameter. Provide a result parameter in the procedure definition.
ERROR TYPE:	illegal parameter profile
message	`Error... The procedure` *`procedure_name`* `of SQL statement type` *`procedures_SQL_type`* `must have parameters.`
correction	This procedure corresponds to an SQL statement that always expects parameters. Re-define this procedure as a procedure that has parameters.
ERROR TYPE:	invalid concrete parameter
message	`Error...` *`Concrete_parameter_name`* `in procedure` *`procedure_name`* `is not a valid parameter to send to a concrete interface.`

correction	All parameters passed to the concrete interface must have been passed to the abstract interface first (with the exception of indicator variables or the "SQLCODE" parameter). Either remove the illegal concrete parameter or add a corresponding abstract parameter first.
ERROR TYPE:	illegal record type
message	`Error... Record types are only allowed to be sent to abstract procedures which represent select, fetch, or insert SQL statements.`
correction	Replace the record type parameter with parameter(s) appropriate to the procedure kind.
ERROR TYPE:	no ops_package
message	`Error...` *`concrete_parameter_name`* `in procedure` *`procedure_name`* `is a null bearing type with no ops package instantiated for it..therefore it cannot be re-assigned to a variable of a concrete type.`
correction	Either redefine the domain definition to include both null and not null bearing types and the ops package, or never reassign the null bearing type.

APPENDIX D
SAMPLE PROGRAMS

D.1 SAMPLE PROGRAMS OVERVIEW

In order to provide more comprehensive examples of domain and interface definition, generation and use, this appendix contains two sets of sample programs. The first set in Section D.2 shows a domain definition procedure, Every_Kind_Domain_View, that defines a domain and subtype of every possible SQL type. All products generated by this procedure are also shown. Section D.3 shows examples of the entire binding process, from domain definition to a large-scale application. This Ada application imports three abstract domain packages definition and two abstract interfaces. The two abstract interfaces contain between them 8 row record definitions and 28 procedures.

D.2 DOMAIN DEFINITION AND GENERATION FOR ALL SQL TYPES

Table D-1. Every_Kind_Domain_View - Domain Definition Procedure

```
-- This procedure shows domain definition for all domain types and
-- subtypes with one example for each sql type possible.
-- SQL types are char, int, smallint, double_precision, enumeration,
-- real, decimal.

with abstract_domain_generator;
with generator_support; use generator_support;

procedure every_kind_domain_view is

   type package_names is (suppliers_def_pkg, parts_def_pkg);
   package dom_packs is new abstract_domain_generator(package_names);

begin
   declare
      type doms is (sname, sno, sno_over_50, colors, colors_not_red,
                    tax_range, tax_over_33, status, high_status);

      package domain_1 is new dom_packs.generate_domain_package
                                (suppliers_def_pkg, doms);

      -- declares domain of type CHAR.  No CHAR subtypes are allowed.
      package first is new domain_1.generate_int_domain
                    (sname, char, null_and_not_null, 1, 15);

      -- declares domain of type INT.
      package second is new domain_1.generate_int_domain
                    (sno, int, null_and_not_null, 1, 100);

      -- declares INT subtype.
      package third is new domain_1.generate_subint_domain
            (sno_over_50, sno, int, null_and_not_null, 51, 100);
   -- declares domain of type ENUMERATION
   type vals is (red, white, blue);
   package fourth is new domain_1.generate_enum_domain
                     (vals, colors, enumeration, contains_null);

   -- declares ENUMERATION subtype
   package fifth is new domain_1.generate_subenum_domain
                     (colors_not_red, colors, enumeration,
                     contains_null, "white", "blue");

   -- declares domain of type REAL
   package sixth is new domain_1.generate_flt_domain
                     (tax_range, real, not_null, 0.000, 100.000);

   -- declares REAL subtype
   package seventh is new domain_1.generate_subflt_domain
         (tax_over_33, tax_range, real, not_null, 33.001, 100.000);

   -- declares domain of type SMALLINT
   package eighth is new domain_1.generate_int_domain
                     (status, smallint, contains_null, 0, 3);

   -- declares SMALLINT subtype
   package nine is new domain_1.generate_subint_domain
         (high_status, status, smallint, contains_null, 3, 3);

   begin
      domain_1.start_generation;
   end;

declare
   type doms is (pno, price, mid_price, weight, light_weight);

   package domain_2 is new dom_packs.generate_domain_package
                                        (parts_def_pkg, doms);

   -- declares domain of type INT
   package first is new domain_2.generate_int_domain
                               (pno, int, not_null, 0, 550);

   -- declares domain of type DECIMAL
   package second is new domain_2.generate_flt_domain
         (price, decimal, null_and_not_null, 0.00, 1000.00);

   -- declares DECIMAL subtype
   package third is new domain_2.generate_subflt_domain
    (mid_price, price, decimal, null_and_not_null, 300.33, 600.66);

   -- declares domain of type DOUBLE_PRECISION
   package fourth is new domain_2.generate_flt_domain
         (weight, double_precision, contains_null, 1.456, 45.567);

   -- declares DOUBLE_PRECISION subtype
   package fifth is new domain_2.generate_subflt_domain
                     (light_weight, weight, double_precision,
                                 contains_null, 1.456, 10.600);

   begin
```

```
         domain_2.start_generation;
      end;

   dom_packs.generate_base_specific;

end every_kind_domain_view;
```

Table D-2. Parts_Def_Pkg - 1st Abstract Domain Package Specification

```
with sql_int_pkg;
use sql_int_pkg;
with sql_double_precision_pkg;
use sql_double_precision_pkg;
with sql_real_pkg;
use sql_real_pkg;
package parts_def_pkg is

   type pno_not_null is new sql_int_not_null
                                    range 0..550;

   type price_not_null is new sql_real_not_null
                                    range 0.00000..1000.00000;
   type price_type is new sql_real;
   package price_ops is new
            sql_real_ops(price_type, price_not_null);

   subtype mid_price_not_null is price_not_null
                                    range 300.32999..600.65997;
   subtype mid_price_type is price_type;
   package mid_price_ops is new
            sql_real_ops(mid_price_type, mid_price_not_null);

   type weight_type is new sql_double_precision;

   subtype light_weight_type is weight_type;

end parts_def_pkg;
```

Table D-3. Suppliers_Def_Pkg - 2nd Abstract Domain Package Specification

```
with sql_char_pkg;
use sql_char_pkg;
with sql_int_pkg;
use sql_int_pkg;
with sql_smallint_pkg;
use sql_smallint_pkg;
with sql_enumeration_pkg;
with sql_real_pkg;
use sql_real_pkg;
package suppliers_def_pkg is

   type snamenn_base is new sql_char_pkg.sql_char_not_null;
   subtype sname_not_null is snamenn_base (1..15);
   type sname_base is new sql_char_pkg.sql_char;
   subtype sname_type is sname_base (sname_not_null'length);
   package sname_ops is new
            sql_char_pkg.sql_char_ops(sname_base, snamenn_base);
```

```
    type sno_not_null is new sql_int_not_null
                                     range 1..100;
    type sno_type is new sql_int;
    package sno_ops is new
             sql_int_ops(sno_type, sno_not_null);

    subtype sno_over_50_not_null is sno_not_null
                                     range 51..100;
    subtype sno_over_50_type is sno_type;
    package sno_over_50_ops is new
             sql_int_ops(sno_over_50_type, sno_over_50_not_null);

    type colors_not_null is (red, white, blue) ;
    package colors_ops is new
             sql_enumeration_pkg(colors_not_null);
    type colors_type is new colors_ops.sql_enumeration;

    subtype colors_not_red_not_null is colors_not_null
                                     range white..blue;
    package colors_not_red_ops is new
             sql_enumeration_pkg(colors_not_red_not_null);
    type colors_not_red_type is new
             colors_not_red_ops.sql_enumeration;

    type tax_range_not_null is new sql_real_not_null
                                     range 0.00000..100.00000;

    subtype tax_over_33_not_null is tax_range_not_null
                                     range 33.00100..100.00000;

    type status_type is new sql_smallint;

    subtype high_status_type is status_type;

 end suppliers_def_pkg;
```

Table D-4. Base_Specific_Domains - Domain Description Specification

```
 with generator_support;
 use generator_support;
 package base_specific_domains is

    type base_specific_domain_types is (sname_not_null, sname_type,
             sno_not_null, sno_type, sno_over_50_not_null,
             sno_over_50_type, colors_type, colors_not_red_type,
             tax_range_not_null, tax_over_33_not_null,
             status_type, high_status_type, pno_not_null,
             price_not_null, price_type, mid_price_not_null,
             mid_price_type, weight_type, light_weight_type,
             null_domain_type);

    type corresponding_concrete_types is (sname_not_null_char_15,
             sname_type_char_15, sno_not_null_int, sno_type_int,
             sno_over_50_not_null_int, sno_over_50_type_int,
             colors_type_enumeration, colors_not_red_type_enumeration,
             tax_range_not_null_real, tax_over_33_not_null_real,
             status_type_smallint, high_status_type_smallint,
          pno_not_null_int, price_not_null_real, price_type_real,
          mid_price_not_null_real, mid_price_type_real,
          weight_type_double_precision,
          light_weight_type_double_precision);

   type valid_domain_names is (suppliers_def_pkg, parts_def_pkg);

   type ops_packages is (sname_ops, sno_ops, sno_over_50_ops,
          price_ops, mid_price_ops);

   longest_enum_value : constant integer :=  5;

   type domain_package_array is array (positive range <>) of
          valid_domain_names;

 end base_specific_domains;
```

D.3. LARGE-SCALE BINDING PROCESS

Table D-5. DOCTORS.SQLMOD - SQL Module

```
 -- This SQL module provides the SQL procedures needed by the
 -- doctors.ada program.

 -----------------------------------------------------------------
 -- Header Information Section
 -----------------------------------------------------------------
 module       doctors                    -- module name
 language     ada                        -- language of calling program
 authorization  physician_reference     -- provides default db handle

 -----------------------------------------------------------------
 -- Declare Statements Section
 -----------------------------------------------------------------
 declare schema filename 'physician_reference'   -- Declaration of the
                                                  -- database

 declare dr_row cursor for
    select p.dr_name, p.dr_address, p.phone_number, p.office_hours,
                     p.area, p.specialty, p.bedside_manner
    from physician p
    where p.dr_name = in_dr_name

 declare dr_hosp cursor for
    select p.dr_name, p.dr_address, p.phone_number
    from physician p, dr_hosp d
    where d.dr_name = p.dr_name and d.hosp_name = in_hosp_name

 declare spec_area cursor for
    select p.dr_name, p.dr_address, p.phone_number
    from physician p
    where p.specialty = in_spec and p.area = in_area

 declare doc_ins cursor for
    select p.dr_name, p.dr_address, p.phone_number
    from physician p, dr_ins d
    where p.dr_name = d.dr_name and d.in_name = in_ins_name
```

```
declare area_ins cursor for
   select p.dr_name, p.dr_address, p.phone_number
   from physician p, dr_ins d
   where p.dr_name = d.dr_name and in_ins_name = d.in_name and
            in_area = p.area

----------------------------------------------------------------------
-- Procedure Section
----------------------------------------------------------------------
-- This procedure uses the executable form for starting a transaction
procedure set_transaction
 sqlcode;

 set transaction read write;

-- This procedure opens the cursor that has been declared for the
-- physician table
procedure open_dr_row
   in_dr_name char (25)
   sqlcode;

   open dr_row;

-- This procedure closes the cursor
procedure close_dr_row
   sqlcode;

   close dr_row;

-- This procedure fetches the data from the opened dr_row cursor
procedure fetch_doctor
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   in_office_hours char (20) in_office_ind smallint
   in_area char(5) in_area_ind smallint
   in_specialty char (11) in_specialty_ind smallint
   in_bedside_manner int in_bedside_ind smallint
   sqlcode;

   fetch dr_row into in_dr_name,
                        in_dr_address,
                        in_phone_number indicator in_phone_ind,
                        in_office_hours  indicator in_office_ind,
                        in_area  indicator in_area_ind,
                        in_specialty indicator in_specialty_ind,
                        in_bedside_manner indicator in_bedside_ind;

-- This procedure updates the doctor information by incrementing
-- the bedside manner
procedure increment_bedside_manner
   sqlcode;

   update physician
   set bedside_manner = bedside_manner + 1
   where current of dr_row;
```

```
--This procedure adds a new insurance company to the database
procedure insert_insurance
   in_name char(20)
   deduc real
   ov real
   drug real
   sqlcode;

   insert into insurance
   values(in_name,deduc,ov,drug);

--This procedure adds a new hospital to the database
procedure insert_hosp
   in_name char(20)
   in_address char (20)
   sqlcode;

   insert into hospital
   values(in_name,in_address);

--This procedure adds a new dr to the database
procedure insert_dr
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   in_office_hours char (20) in_off_hrs_ind smallint
   in_area char(5) in_area_ind smallint
   in_specialty char (11) in_spec_ind smallint
   in_bedside_manner int in_bed_man_ind smallint
   sqlcode;

   insert into physician
   values(in_dr_name,in_dr_address,
            in_phone_number in_phone_ind,
            in_office_hours in_off_hrs_ind,
            in_area in_area_ind,
            in_specialty in_spec_ind,
            in_bedside_manner in_bed_man_ind);

--This procedure adds a new doctor & insurance pair to the
-- dr_ins table
procedure insert_dr_ins
   in_dr_name char (25)
   in_ins_name char (20)
   sqlcode;

   insert into dr_ins values (in_dr_name, in_ins_name);

--This procedure adds a new doctor & hospital pair to the
-- dr_hosp table
procedure insert_dr_hosp
   in_dr_name char (25)
   in_hosp_name char (20)
   sqlcode;

   insert into dr_hosp values (in_dr_name, in_hosp_name);

--This procedure tells doctors at a certain hospital
```

```
procedure open_dr_hosp
   in_hosp_name char (20)
   sqlcode;

   open dr_hosp;

-- This procedure fetches a row from the dr_hosp cursor
procedure fetch_dr_hosp
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   sqlcode;

   fetch dr_hosp into in_dr_name,
                      in_dr_address,
                      in_phone_number indicator in_phone_ind;

-- This procedure closes the cursor
procedure close_dr_hosp
   sqlcode;

   close dr_hosp;

--This procedure deletes a doctor from the dr table
procedure delete_dr_in_dr
   in_dr_name char (25)
   sqlcode;

   delete from physician
   where dr_name = in_dr_name;

--This procedure deletes a doctor from insurance table
procedure delete_dr_in_ins
   in_dr_name char (25)
   sqlcode;

   delete from dr_ins
   where dr_name = in_dr_name;

--This procedure deletes a doctor from the hospital table
procedure delete_dr_in_hosp
   in_dr_name char (25)
   sqlcode;

   delete from dr_hosp
   where dr_name = in_dr_name;

--This procedure selects all doctors w/ a certain
-- specialty within a certain area
procedure open_spec_area
   in_spec char (11)
   in_area char (5)
   sqlcode;

   open spec_area;

--This procedure will fetch a row from the spec_area cursor
procedure fetch_spec_area
```

```
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   sqlcode;

   fetch spec_area into in_dr_name,
                        in_dr_address,
                        in_phone_number indicator in_phone_ind;

-- This procedure closes the cursor
procedure close_spec_area
   sqlcode;

   close spec_area;

--This procedure selects doctors with certain insurance
procedure open_doc_ins
   in_ins_name char (20)
   sqlcode;

   open doc_ins;

--This procedure will fetch a row from the doc_ins cursor
procedure fetch_doc_ins
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   sqlcode;

   fetch doc_ins into in_dr_name,
                      in_dr_address,
                      in_phone_number indicator in_phone_ind;

-- This procedure closes the cursor
procedure close_doc_ins
   sqlcode;

   close doc_ins;

--This procedure will update information about a doctor
procedure update_doctor
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   in_office_hours char (20) in_office_ind smallint
   in_area char(5) in_area_ind smallint
   in_specialty char (11) in_specialty_ind smallint
   in_bedside_manner int in_bedside_ind smallint
   sqlcode;


   update physician
   set dr_address = in_dr_address,
   phone_number = in_phone_number in_phone_ind,
   office_hours = in_office_hours in_office_ind,
   area = in_area in_area_ind,
   specialty = in_specialty in_specialty_ind,
   bedside_manner = in_bedside_manner in_bedside_ind
   where dr_name = in_dr_name;

--This procedure selects doctors within a specific
-- area that honor a specific insurance
procedure open_area_ins
   in_ins_name char (20)
   in_area char(5)
   sqlcode;

   open area_ins;

--This procedure will fetch a row from the area_ins cursor
procedure fetch_area_ins
   in_dr_name char (25)
   in_dr_address char (25)
   in_phone_number char (11) in_phone_ind smallint
   sqlcode;

   fetch area_ins into in_dr_name,
                       in_dr_address,
                       in_phone_number indicator in_phone_ind;

-- This procedure closes the cursor
procedure close_area_ins
   sqlcode;

   close area_ins;


-- This procedure commits the transaction
procedure commit_transaction
   sqlcode;

   commit;

-- This procedure rolls back the transaction
procedure rollback_transaction
   sqlcode;

   rollback;
```

Table D-6. Doctors_Domview - Domain Definition Procedure

```
with abstract_domain_generator;
with generator_support; use generator_support;

procedure doctors_domview is

   type PACK_NAMES is (doctors_def_enum_pkg, hospital_def_pkg,
                                 insurance_def_pkg);
   package DOM_PACKS is new abstract_domain_generator (PACK_NAMES);

begin
   declare
      type DOMS is (dr_name, dr_address, phone_number, office_hours,
                              area, specialty, bedside_manner);
      package DOMAIN_1 is new DOM_PACKS.generate_domain_package
                              (doctors_def_enum_pkg , DOMS);
```

```
         package FIRST is new DOMAIN_1.generate_int_domain
            (dr_name, char, not_null, 1, 25);
         package SECOND is new DOMAIN_1.generate_int_domain
            (dr_address, char, not_null, 1, 25);
         package THIRD is new DOMAIN_1.generate_int_domain
            (phone_number, char, null_and_not_null, 1, 11);
         package FOURTH is new DOMAIN_1.generate_int_domain
            (office_hours, char, null_and_not_null, 1, 20);
         type areas is (north, south, east, west);
         package FIFTH is new DOMAIN_1.generate_enum_domain
            (areas, area, enumeration, null_and_not_null);
         package SIXTH is new DOMAIN_1.generate_int_domain
            (specialty, char, null_and_not_null, 1, 11);
         package SEVENTH is new DOMAIN_1.generate_int_domain
            (bedside_manner, int, null_and_not_null, 1, 10);
      begin
         DOMAIN_1.start_generation;
      end;

   declare
      type DOMS is (ins_name, deductible, ov_copay, drug_copay);
      package DOMAIN_2 is new DOM_PACKS.generate_domain_package
                                 (insurance_def_pkg , DOMS);
         package FIRST is new DOMAIN_2.generate_int_domain
            (ins_name, char, not_null, 1, 20);
         package SECOND is new DOMAIN_2.generate_flt_domain
            (deductible, real, not_null, 0.00, 2500.00);
         package THIRD is new DOMAIN_2.generate_flt_domain
            (ov_copay, real, not_null, 0.00, 25.00);
         package FOURTH is new DOMAIN_2.generate_flt_domain
            (drug_copay, real, not_null, 0.00, 20.00);
      begin
         DOMAIN_2.start_generation;
      end;

   declare
      type DOMS is (hosp_name, hosp_address);
      package DOMAIN_3 is new DOM_PACKS.generate_domain_package
                                 (hospital_def_pkg , DOMS);
         package FIRST is new DOMAIN_3.generate_int_domain
            (hosp_name, char, not_null, 1, 20);
         package SECOND is new DOMAIN_3.generate_int_domain
            (hosp_address, char, not_null, 1, 20);
      begin
         DOMAIN_3.start_generation;
      end;

   DOM_PACKS.generate_base_specific;

end doctors_domview;
```

Table D-7. Doctors_Def_Enum_Pkg - 1st Abstract Domain Package Specification

```
with sql_char_pkg;
use sql_char_pkg;
with sql_int_pkg;
use sql_int_pkg;
with sql_enumeration_pkg;
```

```
package doctors_def_enum_pkg is

   type dr_namenn_base is new sql_char_pkg.sql_char_not_null;
   subtype dr_name_not_null is dr_namenn_base (1..25);

   type dr_addressnn_base is new sql_char_pkg.sql_char_not_null;
   subtype dr_address_not_null is dr_addressnn_base (1..25);

   type phone_numbernn_base is new sql_char_pkg.sql_char_not_null;
   subtype phone_number_not_null is phone_numbernn_base (1..11);
   type phone_number_base is new sql_char_pkg.sql_char;
   subtype phone_number_type is phone_number_base
                                    (phone_number_not_null'length);

   package phone_number_ops is new
            sql_char_pkg.sql_char_ops(phone_number_base,
                                                phone_numbernn_base);

   type office_hoursnn_base is new sql_char_pkg.sql_char_not_null;
   subtype office_hours_not_null is office_hoursnn_base (1..20);
   type office_hours_base is new sql_char_pkg.sql_char;
   subtype office_hours_type is office_hours_base
                                    (office_hours_not_null'length);

   package office_hours_ops is new
            sql_char_pkg.sql_char_ops(office_hours_base,
                                                office_hoursnn_base);

   type area_not_null is (north, south, east, west);
   package area_ops is new
            sql_enumeration_pkg(area_not_null);

   type specialtynn_base is new sql_char_pkg.sql_char_not_null;
   subtype specialty_not_null is specialtynn_base (1..11);
   type specialty_base is new sql_char_pkg.sql_char;
   subtype specialty_type is specialty_base
                                    (specialty_not_null'length);
   package specialty_ops is new
            sql_char_pkg.sql_char_ops(specialty_base,
                                                specialtynn_base);

   type bedside_manner_not_null is new sql_int_not_null
                       range 1..10;
   type bedside_manner_type is new sql_int;
   package bedside_manner_ops is new
            sql_int_ops(bedside_manner_type, bedside_manner_not_null);

end doctors_def_enum_pkg;
```

Table D-8. Hospital_Def_Pkg - 2nd Abstract Domain Package Specification

```
with sql_char_pkg;
use sql_char_pkg;
package hospital_def_pkg is

   type hosp_namenn_base is new sql_char_pkg.sql_char_not_null;
   subtype hosp_name_not_null is hosp_namenn_base (1..20);
```

```
    type hosp_addressnn_base is new sql_char_pkg.sql_char_not_null;
    subtype hosp_address_not_null is hosp_addressnn_base (1..20);

end hospital_def_pkg;
```

Table D-9. Insurance_Def_Pkg - 3rd Abstract Domain Package Specification

```
with sql_char_pkg;
use sql_char_pkg;
with sql_real_pkg;
use sql_real_pkg;
package insurance_def_pkg is

    type ins_namenn_base is new sql_char_pkg.sql_char_not_null;
    subtype ins_name_not_null is ins_namenn_base (1..20);

    type deductible_not_null is new sql_real_not_null
    range 0.00000..2500.00000;

    type ov_copay_not_null is new sql_real_not_null
    range 0.00000..25.00000;

    type drug_copay_not_null is new sql_real_not_null
    range 0.00000..20.00000;

end insurance_def_pkg;
```

Table D-10. Base_Specific_Domains - Domain Description Specification

```
with generator_support;
use generator_support;
package base_specific_domains is

    type base_specific_domain_types is (dr_name_not_null,
             dr_address_not_null, phone_number_not_null,
             phone_number_type,office_hours_not_null,
             office_hours_type, area_not_null, area_type,
             specialty_not_null, specialty_type,
             bedside_manner_not_null, bedside_manner_type,
             ins_name_not_null, deductible_not_null,
             ov_copay_not_null, drug_copay_not_null,
             hosp_name_not_null, hosp_address_not_null,
             null_domain_type);

    type corresponding_concrete_types is (dr_name_not_null_char_25,
             dr_address_not_null_char_25,phone_number_not_null_char_11,
             phone_number_type_char_11, office_hours_not_null_char_20,
             office_hours_type_char_20, area_not_null_enumeration,
             area_type_enumeration, specialty_not_null_char_11,
             specialty_type_char_11, bedside_manner_not_null_int,
             bedside_manner_type_int, ins_name_not_null_char_20,
             deductible_not_null_real, ov_copay_not_null_real,
             drug_copay_not_null_real, hosp_name_not_null_char_20,
             hosp_address_not_null_char_20);

    type valid_domain_names is (doctors_def_enum_pkg, hospital_def_pkg,
                                                     insurance_def_pkg);
```

```
   type ops_packages is (phone_number_ops, office_hours_ops, area_ops,
                         specialty_ops, bedside_manner_ops);

   longest_enum_value : constant integer :=  5;

   type domain_package_array is array (positive range <>) of
                                               valid_domain_names;

end base_specific_domains;
```

Table D-11. Doctors_Intview_Enum - 1st Interface Definition Procedure

```
with base_specific_domains;
use base_specific_domains;
with abstract_interface_generator;
with generator_support; use generator_support;

procedure doctors_intview_enum is

   type record_names is (doctor_record, insurance_record,
                         hospital_record, dr_ins_record,
                         dr_hosp_record);
   type proc_names is (open_dr_row, close_dr_row, fetch_doctor,
                         insert_insurance, insert_hosp,
                         insert_dr, insert_dr_ins, insert_dr_hosp,
                         delete_dr_in_dr, delete_dr_in_ins,
                         delete_dr_in_hosp, update_doctor,
                         commit_transaction,. rollback_transaction);

   type ok_errors is (cursor_already_open, cursor_not_open, not_found,
                         duplicate_value, fetch_not_done);

   package my_interface is new abstract_interface_generator
            ("concrete_interface_enum", "abstract_interface_enum",
            "dbms_specific_enum", (1 => doctors_def_enum_pkg,
            2 => insurance_def_pkg, 3 => hospital_def_pkg),
            record_names, proc_names, ok_errors,
            (1=> 1001, 2=> -501, 3=> 100, 4=> -803,  5=> -508));
   use my_interface;

   begin
      declare
         type rec_components is (dname, daddress, dphone, doff_hrs,
                                 darea, dspec, dbed_man);
         package rec1 is new record_generator (rec_components,
                                               doctor_record);
            package com_11 is new rec1.component_generator
                                 (dname, dr_name_not_null);
            package com_12 is new rec1.component_generator
                                 (daddress, dr_address_not_null);
            package com_13 is new rec1.component_generator
                                 (dphone, phone_number_type);
            package com_14 is new rec1.component_generator
                                 (doff_hrs, office_hours_type);
            package com_15 is new rec1.component_generator
                                 (darea, area_type);
         package com_16 is new rec1.component_generator
                                  (dspec, specialty_type);
         package com_17 is new rec1.component_generator
                                  (dbed_man, bedside_manner_type);
      begin
         rec1.generate_record;
      end;

   declare
      type rec_components is (iname, ided, iov_co, idr_co);
      package rec2 is new record_generator (rec_components,
                                  insurance_record);
         package com_21 is new rec2.component_generator
                                  (iname, ins_name_not_null);
         package com_22 is new rec2.component_generator
                                  (ided, deductible_not_null);
         package com_23 is new rec2.component_generator
                                  (iov_co, ov_copay_not_null);
         package com_24 is new rec2.component_generator
                                  (idr_co, drug_copay_not_null);
      begin
         rec2.generate_record;
      end;

   declare
      type rec_components is (hname, haddress);
      package rec3 is new record_generator (rec_components,
                                              hospital_record);
         package com_31 is new rec3.component_generator
                                  (hname, hosp_name_not_null);
         package com_32 is new rec3.component_generator
                                  (haddress, hosp_address_not_null);
      begin
         rec3.generate_record;
      end;

   declare
      type rec_components is (dname, iname);
      package rec5 is new record_generator (rec_components,
                                              dr_ins_record);
         package com_51 is new rec5.component_generator
                                  (dname, dr_name_not_null);
         package com_52 is new rec5.component_generator
                                  (iname, ins_name_not_null);
      begin
         rec5.generate_record;
      end;

   declare
      type rec_components is (dname, hname);
      package rec6 is new record_generator (rec_components,
                                              dr_hosp_record);
         package com_61 is new rec6.component_generator
                                  (dname, dr_name_not_null);
         package com_62 is new rec6.component_generator
                                  (hname, hosp_name_not_null);
      begin
         rec6.generate_record;
      end;

   declare
      type params is (in_dr_name, result);
      type params_conc is (in_dr_name);
      package procedure1 is new procedure_with_parameters_generator
                      (procedure_name => open_dr_row,
                      parameters => params,
                      sql_statement_type => open,
                      sql_module_procedure_name => "open_dr_row",
                      params_to_concrete_procedure => params_conc,
                      valid_errors => (1 => cursor_already_open));
      use procedure1;
         package param1 is new params_of_domain_type_generator
                                (in_dr_name, dr_name_not_null);
         package param2 is new params_of_error_conditions_generator
                                (result);
      begin
         procedure1.generate_procedure;
      end;

   declare
      package procedure2 is new
         procedure_without_parameters_generator
                      (procedure_name => close_dr_row,
                      sql_statement_type => close,
                      sql_module_procedure_name => "close_dr_row");
      begin
         procedure2.generate_procedure;
      end;

   declare
      type params is (dr_record, result);
      type params_conc is (dname, daddress, dphone, doff_hrs,
                                darea, dspec, dbed_man);
      package procedure3 is new procedure_with_parameters_generator
                      (procedure_name => fetch_doctor,
                      parameters =>  params,
                      sql_statement_type => fetch,
                      sql_module_procedure_name => "fetch_doctor",
                      params_to_concrete_procedure => params_conc,
                      valid_errors => (1 => cursor_not_open,
                                       2 => not_found));
      use procedure3;
         package param1 is new params_of_record_type_generator
                                (dr_record, doctor_record);
         package param2 is new params_of_error_conditions_generator
                                (result);
      begin
         procedure3.generate_procedure;
      end;

   declare
      type params is (ins_record, result);
      type params_conc is (iname, ided, iov_co, idr_co, result);
      package procedure5 is new procedure_with_parameters_generator
                      (procedure_name => insert_insurance,
                      parameters =>  params,
                    sql_statement_type => insert_values,
                    sql_module_procedure_name =>
                                            "insert_insurance",
                    params_to_concrete_procedure => params_conc,
                    valid_errors => (1 => duplicate_value));
      use procedure5;
         package param1 is new params_of_record_type_generator
                                 (ins_record, insurance_record);
         package param2 is new params_of_error_conditions_generator
                                 (result);
      begin
         procedure5.generate_procedure;
      end;

   declare
      type params is (hosp_record, result);
      type params_conc is (hname, haddress, result);
      package procedure6 is new procedure_with_parameters_generator
                    (procedure_name => insert_hosp,
                    parameters =>  params,
                    sql_statement_type => insert_values,
                    sql_module_procedure_name => "insert_hosp",
                    params_to_concrete_procedure => params_conc,
                    valid_errors => (1 => duplicate_value));
      use procedure6;
         package param1 is new params_of_record_type_generator
                                 (hosp_record, hospital_record);
         package param2 is new params_of_error_conditions_generator
                                 (result);
      begin
         procedure6.generate_procedure;
      end;

   declare
      type params is (dr_record, result);
      type params_conc is (dname, daddress, dphone, doff_hrs,
                                 darea, dspec, dbed_man, result);
      package procedure7 is new procedure_with_parameters_generator
                    (procedure_name => insert_dr,
                    parameters =>  params,
                    sql_statement_type => insert_values,
                    sql_module_procedure_name => "insert_dr",
                    params_to_concrete_procedure => params_conc,
                    valid_errors => (1 => duplicate_value));
      use procedure7;
         package param1 is new params_of_record_type_generator
                                 (dr_record, doctor_record);
         package param2 is new params_of_error_conditions_generator
                                 (result);
      begin
         procedure7.generate_procedure;
      end;

   declare
      type params is (dr_ins_rec, result);
      type params_conc is (dname, iname, result);
      package procedure8 is new procedure_with_parameters_generator
                    (procedure_name => insert_dr_ins,
                    parameters => params,
                    sql_statement_type => insert_values,
                    sql_module_procedure_name => "insert_dr_ins",
                    params_to_concrete_procedure => params_conc,
                    valid_errors => (1 => duplicate_value));
      use procedure8;
         package param1 is new params_of_record_type_generator
                                (dr_ins_rec, dr_ins_record);
         package param2 is new params_of_error_conditions_generator
                                (result);
      begin
         procedure8.generate_procedure;
      end;

   declare
      type params is (dr_hosp_rec, result);
      type params_conc is (dname, hname, result);
      package procedure9 is new procedure_with_parameters_generator
                    (procedure_name => insert_dr_hosp,
                    parameters => params,
                    sql_statement_type => insert_values,
                    sql_module_procedure_name => "insert_dr_hosp",
                    params_to_concrete_procedure => params_conc,
                    valid_errors =>  (1 => duplicate_value));
      use procedure9;
         package param1 is new params_of_record_type_generator
                                (dr_hosp_rec, dr_hosp_record);
         package param2 is new params_of_error_conditions_generator
                                (result);
      begin
         procedure9.generate_procedure;
      end;

   declare
      type params is (dr_name, result);
      type params_conc is (dr_name, result);
      package procedure13 is new
         procedure_with_parameters_generator
                    (procedure_name => delete_dr_in_dr,
                    parameters => params,
                    sql_statement_type => delete_searched,
                    sql_module_procedure_name =>
                                          "delete_dr_in_dr",
                    params_to_concrete_procedure => params_conc,
                    valid_errors =>  (1 => not_found));
      use procedure13;
         package param1 is new params_of_domain_type_generator
                                (dr_name, dr_name_not_null);
         package param2 is new params_of_error_conditions_generator
                                (result);
      begin
         procedure13.generate_procedure;
      end;

   declare
      type params is (dr_name, result);
      type params_conc is (dr_name, result);
      package procedure14 is new
      procedure_with_parameters_generator
                  (procedure_name => delete_dr_in_ins,
                  parameters => params,
                  sql_statement_type => delete_searched,
                  sql_module_procedure_name =>
                                         "delete_dr_in_ins",
                  params_to_concrete_procedure => params_conc,
                  valid_errors => (1 => not_found));
   use procedure14;
      package param1 is new params_of_domain_type_generator
                           (dr_name, dr_name_not_null);
      package param2 is new params_of_error_conditions_generator
                           (result);
   begin
      procedure14.generate_procedure;
   end;

declare
   type params is (dr_name, result);
   type params_conc is (dr_name, result);
   package procedure15 is new
      procedure_with_parameters_generator
                  (procedure_name => delete_dr_in_hosp,
                  parameters => params,
                  sql_statement_type => delete_searched,
                  sql_module_procedure_name =>
                                         "delete_dr_in_hosp",
                  params_to_concrete_procedure => params_conc,
                  valid_errors => (1 => not_found));
   use procedure15;
      package param1 is new params_of_domain_type_generator
                           (dr_name, dr_name_not_null);
      package param2 is new params_of_error_conditions_generator
                           (result);
   begin
      procedure15.generate_procedure;
   end;

declare
   package procedure19 is new
      procedure_without_parameters_generator
                  (procedure_name => commit_transaction,
                  sql_statement_type => commit,
                  sql_module_procedure_name =>
                                         "commit_transaction");
   begin
      procedure19.generate_procedure;
   end;

declare
   package procedure20 is new
      procedure_without_parameters_generator
                  (procedure_name => rollback_transaction,
                  sql_statement_type => rollback,
                  sql_module_procedure_name =>
                           "rollback_transaction");
   begin
      procedure20.generate_procedure;
      end;

    declare
      type params is (doc_name, doc_address, doc_phone,
                                 doc_off_hrs, doc_area, doc_spec,
                                 doc_bed_man, result);
      type params_conc is (doc_name, doc_address, doc_phone,
                                 doc_off_hrs, doc_area,
                                 doc_spec, doc_bed_man, result);
      package procedure27 is new
         procedure_with_parameters_generator
                      (procedure_name => update_doctor,
                      parameters => params,
                      sql_statement_type => update_searched,
                      sql_module_procedure_name => "update_doctor",
                      params_to_concrete_procedure => params_conc,
                      valid_errors => (1 => duplicate_value));
      use procedure27;
         package param1 is new params_of_domain_type_generator
                                 (doc_name, dr_name_not_null);
         package param2 is new params_of_domain_type_generator
                                 (doc_address,dr_address_not_null);
         package param3 is new params_of_domain_type_generator
                                 (doc_phone, phone_number_type);
         package param4 is new params_of_domain_type_generator
                                 (doc_off_hrs, office_hours_type);
         package param5 is new params_of_domain_type_generator
                                 (doc_area, area_type);
         package param6 is new params_of_domain_type_generator
                                 (doc_spec, specialty_type);
         package param7 is new params_of_domain_type_generator
                                 (doc_bed_man,bedside_manner_type);
         package param8 is new params_of_error_conditions_generator
                                 (result);
      begin
         procedure27.generate_procedure;
      end;

    my_interface.generate_interface;

end doctors_intview_enum;
```

Table D-12. Abstract_Interface_Enum - 1st Abstract Interface Package Specification

```
with DOCTORS_DEF_ENUM_PKG;
use DOCTORS_DEF_ENUM_PKG;
with INSURANCE_DEF_PKG;
use INSURANCE_DEF_PKG;
with HOSPITAL_DEF_PKG;
use HOSPITAL_DEF_PKG;
with sql_standard;
use sql_standard;

package abstract_interface_enum is

   type valid_status_result_type is (CURSOR_ALREADY_OPEN,
                                     CURSOR_NOT_OPEN, NOT_FOUND,
                                     DUPLICATE_VALUE, FETCH_NOT_DONE);
```

```
type DOCTOR_RECORD is record
   DNAME : DR_NAME_NOT_NULL;
   DADDRESS : DR_ADDRESS_NOT_NULL;
   DPHONE : PHONE_NUMBER_TYPE;
   DOFF_HRS : OFFICE_HOURS_TYPE;
   DAREA : AREA_TYPE;
   DSPEC : SPECIALTY_TYPE;
   DBED_MAN : BEDSIDE_MANNER_TYPE;
end record;

type INSURANCE_RECORD is record
   INAME : INS_NAME_NOT_NULL;
   IDED : DEDUCTIBLE_NOT_NULL;
   IOV_CO : OV_COPAY_NOT_NULL;
   IDR_CO : DRUG_COPAY_NOT_NULL;
end record;

type HOSPITAL_RECORD is record
   HNAME : HOSP_NAME_NOT_NULL;
   HADDRESS : HOSP_ADDRESS_NOT_NULL;
end record;

type DR_INS_RECORD is record
   DNAME : DR_NAME_NOT_NULL;
   INAME : INS_NAME_NOT_NULL;
end record;

type DR_HOSP_RECORD is record
   DNAME : DR_NAME_NOT_NULL;
   HNAME : HOSP_NAME_NOT_NULL;
end record;

procedure OPEN_DR_ROW(IN_DR_NAME : in DR_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type);

procedure CLOSE_DR_ROW;


procedure FETCH_DOCTOR(DR_RECORD : in out DOCTOR_RECORD;
                     RESULT : out valid_status_result_type);

procedure INSERT_INSURANCE(INS_RECORD : in INSURANCE_RECORD;
                     RESULT : out valid_status_result_type);

procedure INSERT_HOSP(HOSP_RECORD : in HOSPITAL_RECORD;
                     RESULT : out valid_status_result_type);

procedure INSERT_DR(DR_RECORD : in DOCTOR_RECORD;
                     RESULT : out valid_status_result_type);

procedure INSERT_DR_INS(DR_INS_REC : in DR_INS_RECORD;
                     RESULT : out valid_status_result_type);

procedure INSERT_DR_HOSP(DR_HOSP_REC : in DR_HOSP_RECORD;
                     RESULT : out valid_status_result_type);

procedure DELETE_DR_IN_DR(DR_NAME : in DR_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type);
```

```
    procedure DELETE_DR_IN_INS(DR_NAME : in DR_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type);

    procedure DELETE_DR_IN_HOSP(DR_NAME : in DR_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type);

    procedure COMMIT_TRANSACTION;

    procedure ROLLBACK_TRANSACTION;

    procedure UPDATE_DOCTOR(DOC_NAME : in DR_NAME_NOT_NULL;
                     DOC_ADDRESS : in DR_ADDRESS_NOT_NULL;
                     DOC_PHONE : in PHONE_NUMBER_TYPE;
                     DOC_OFF_HRS : in OFFICE_HOURS_TYPE;
                     DOC_AREA : in AREA_TYPE;
                     DOC_SPEC : in SPECIALTY_TYPE;
                     DOC_BED_MAN : in BEDSIDE_MANNER_TYPE;
                     RESULT : out valid_status_result_type);

end abstract_interface_enum;
```

Table D-13. Abstract_Interface_Enum - 1st Abstract Interface Package Body

```
with sql_communications_pkg, sql_database_error_pkg, conversions,
 concrete_interface_enum;

use sql_communications_pkg, sql_database_error_pkg, conversions;

package body abstract_interface_enum is

   use PHONE_NUMBER_OPS, OFFICE_HOURS_OPS, AREA_OPS, SPECIALTY_OPS,
         BEDSIDE_MANNER_OPS;

   CURSOR_ALREADY_OPEN_VALUE : constant :=  1001;
   CURSOR_NOT_OPEN_VALUE : constant := -501;
   NOT_FOUND_VALUE : constant :=  100;
   DUPLICATE_VALUE_VALUE : constant := -803;
   FETCH_NOT_DONE_VALUE : constant := -508;

   procedure OPEN_DR_ROW(IN_DR_NAME : in DR_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.open_dr_row
                                 (CHAR(IN_DR_NAME), SQLCODE);
      if sqlcode = CURSOR_ALREADY_OPEN_VALUE then
         RESULT := CURSOR_ALREADY_OPEN;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end OPEN_DR_ROW;

   procedure CLOSE_DR_ROW is
   begin
      concrete_interface_enum.close_dr_row (sqlcode);
      if sqlcode /= 0 then
         process_database_error;
      raise sql_database_error;
   end if;
end CLOSE_DR_ROW;

procedure FETCH_DOCTOR(DR_RECORD : in out DOCTOR_RECORD;
                       RESULT : out valid_status_result_type) is
   DPHONE_c : CHAR(1..11) := (others => ' ');
   DPHONE_indic : indicator_type;
   DOFF_HRS_c : CHAR(1..20) := (others => ' ');
   DOFF_HRS_indic : indicator_type;
   DAREA_c : char (1.. 5) := (others => ' ');
   DAREA_indic : indicator_type;
   DSPEC_c : CHAR(1..11) := (others => ' ');
   DSPEC_indic : indicator_type;
   DBED_MAN_c : INT;
   DBED_MAN_indic : indicator_type;
begin
   concrete_interface_enum.fetch_doctor (CHAR(DR_RECORD.DNAME),
                                         CHAR(DR_RECORD.DADDRESS),
                                         DPHONE_c, DPHONE_indic,
                                         DOFF_HRS_c, DOFF_HRS_indic,
                                         DAREA_c, DAREA_indic, DSPEC_c,
                                         DSPEC_indic, DBED_MAN_c,
                                         DBED_MAN_indic, SQLCODE);
   if sqlcode = CURSOR_NOT_OPEN_VALUE then
      RESULT := CURSOR_NOT_OPEN;
   elsif sqlcode = NOT_FOUND_VALUE then
      RESULT := NOT_FOUND;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   else
      assign(DR_RECORD.DPHONE,
         PHONE_NUMBER_base(convert(DPHONE_c, DPHONE_indic)));
      assign(DR_RECORD.DOFF_HRS,
         OFFICE_HOURS_base(convert(DOFF_HRS_c, DOFF_HRS_indic)));
      assign(DR_RECORD.DAREA,
         with_null(AREA_NOT_NULL'value(string(DAREA_c))));
      assign(DR_RECORD.DSPEC,
         SPECIALTY_base(convert(DSPEC_c, DSPEC_indic)));
      assign(DR_RECORD.DBED_MAN,
         BEDSIDE_MANNER_TYPE(convert(DBED_MAN_c, DBED_MAN_indic)));
   end if;
end FETCH_DOCTOR;

procedure INSERT_INSURANCE(INS_RECORD : in INSURANCE_RECORD;
                           RESULT : out valid_status_result_type) is
begin
   concrete_interface_enum.insert_insurance(CHAR(INS_RECORD.INAME),
                                            REAL(INS_RECORD.IDED),
                                            REAL(INS_RECORD.IOV_CO),
                                            REAL(INS_RECORD.IDR_CO),
                                            SQLCODE);
   if sqlcode = DUPLICATE_VALUE_VALUE then
      RESULT := DUPLICATE_VALUE;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
      end if;
   end INSERT_INSURANCE;

      procedure INSERT_HOSP(HOSP_RECORD : in HOSPITAL_RECORD;
                            RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.insert_hosp (CHAR(HOSP_RECORD.HNAME),
      CHAR(HOSP_RECORD.HADDRESS), SQLCODE);
      if sqlcode = DUPLICATE_VALUE_VALUE then
         RESULT := DUPLICATE_VALUE;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end INSERT_HOSP;

   procedure INSERT_DR(DR_RECORD : in DOCTOR_RECORD;
                            RESULT : out valid_status_result_type) is
      DPHONE_c : CHAR(1..11) := (others => ' ');
      DPHONE_indic : indicator_type;
      DOFF_HRS_c : CHAR(1..20) := (others => ' ');
      DOFF_HRS_indic : indicator_type;
      DAREA_c : char (1.. 5) := (others => ' ');
      DAREA_indic : indicator_type;
      DSPEC_c : CHAR(1..11) := (others => ' ');
      DSPEC_indic : indicator_type;
      DBED_MAN_c : INT;
      DBED_MAN_indic : indicator_type;
   begin
      if is_null(DR_RECORD.DPHONE)
         then DPHONE_indic := -1;
      else DPHONE_indic := 0;
         DPHONE_c := char(without_null_base(DR_RECORD.DPHONE));
      end if;
      if is_null(DR_RECORD.DOFF_HRS)
         then DOFF_HRS_indic := -1;
      else DOFF_HRS_indic := 0;
         DOFF_HRS_c := char(without_null_base(DR_RECORD.DOFF_HRS));
      end if;
      if is_null(DR_RECORD.DAREA)
         then DAREA_indic := -1;
      else DAREA_indic := 0;
         DAREA_c :=
         char(AREA_NOT_NULL'image(without_null(DR_RECORD.DAREA)));
      end if;
      if is_null(DR_RECORD.DSPEC)
         then DSPEC_indic := -1;
      else DSPEC_indic := 0;
         DSPEC_c := char(without_null_base(DR_RECORD.DSPEC));
      end if;
      if is_null(DR_RECORD.DBED_MAN)
         then DBED_MAN_indic := -1;
      else DBED_MAN_indic := 0;
         DBED_MAN_c := INT(without_null_base(DR_RECORD.DBED_MAN));
      end if;
      concrete_interface_enum.insert_dr (CHAR(DR_RECORD.DNAME),
                                         CHAR(DR_RECORD.DADDRESS),
                                         DPHONE_c, DPHONE_indic,
                                   DOFF_HRS_c, DOFF_HRS_indic,
                                   DAREA_c, DAREA_indic, DSPEC_c,
                                   DSPEC_indic, DBED_MAN_c,
                                   DBED_MAN_indic, SQLCODE);
      if sqlcode = DUPLICATE_VALUE_VALUE then
         RESULT := DUPLICATE_VALUE;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end INSERT_DR;

   procedure INSERT_DR_INS(DR_INS_REC : in DR_INS_RECORD;
                    RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.insert_dr_ins (CHAR(DR_INS_REC.DNAME),
                                   CHAR(DR_INS_REC.INAME), SQLCODE);
      if sqlcode = DUPLICATE_VALUE_VALUE then
         RESULT := DUPLICATE_VALUE;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end INSERT_DR_INS;

   procedure INSERT_DR_HOSP(DR_HOSP_REC : in DR_HOSP_RECORD;
                    RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.insert_dr_hosp (CHAR(DR_HOSP_REC.DNAME),
                                   CHAR(DR_HOSP_REC.HNAME), SQLCODE);
      if sqlcode = DUPLICATE_VALUE_VALUE then
         RESULT := DUPLICATE_VALUE;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end INSERT_DR_HOSP;

   procedure DELETE_DR_IN_DR(DR_NAME : in DR_NAME_NOT_NULL;
                    RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.delete_dr_in_dr(CHAR(DR_NAME),
                                           SQLCODE);
      if sqlcode = NOT_FOUND_VALUE then
         RESULT := NOT_FOUND;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end DELETE_DR_IN_DR;

   procedure DELETE_DR_IN_INS(DR_NAME : in DR_NAME_NOT_NULL;
                    RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.delete_dr_in_ins (CHAR(DR_NAME),
                                           SQLCODE);
      if sqlcode = NOT_FOUND_VALUE then
         RESULT := NOT_FOUND;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end DELETE_DR_IN_INS;

   procedure DELETE_DR_IN_HOSP(DR_NAME : in DR_NAME_NOT_NULL;
                              RESULT : out valid_status_result_type) is
   begin
      concrete_interface_enum.delete_dr_in_hosp (CHAR(DR_NAME),
                                                 SQLCODE);
      if sqlcode = NOT_FOUND_VALUE then
         RESULT := NOT_FOUND;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end DELETE_DR_IN_HOSP;

   procedure COMMIT_TRANSACTION is
   begin
      concrete_interface_enum.commit_transaction (sqlcode);
      if sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end COMMIT_TRANSACTION;

   procedure ROLLBACK_TRANSACTION is
   begin
      concrete_interface_enum.rollback_transaction (sqlcode);
      if sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end ROLLBACK_TRANSACTION;

   procedure UPDATE_DOCTOR(DOC_NAME : in DR_NAME_NOT_NULL;
                           DOC_ADDRESS : in DR_ADDRESS_NOT_NULL;
                           DOC_PHONE : in PHONE_NUMBER_TYPE;
                           DOC_OFF_HRS : in OFFICE_HOURS_TYPE;
                           DOC_AREA : in AREA_TYPE;
                           DOC_SPEC : in SPECIALTY_TYPE;
                           DOC_BED_MAN : in BEDSIDE_MANNER_TYPE;
                           RESULT : out valid_status_result_type) is

      DOC_PHONE_c:CHAR(PHONE_NUMBER_NOT_NULL'range):= (others => ' ');
      DOC_PHONE_indic : indicator_type;
      DOC_OFF_HRS_c:CHAR(OFFICE_HOURS_NOT_NULL'range):=(others =>' ');
      DOC_OFF_HRS_indic : indicator_type;
      DOC_AREA_c : char (1.. 5) := (others => ' ');
      DOC_AREA_indic : indicator_type;
      DOC_SPEC_c : CHAR(SPECIALTY_NOT_NULL'range) := (others => ' ');
      DOC_SPEC_indic : indicator_type;
      DOC_BED_MAN_c : INT;
      DOC_BED_MAN_indic : indicator_type;
   begin
      if is_null(DOC_PHONE)
```

```
         then DOC_PHONE_indic := -1;
      else DOC_PHONE_indic := 0;
         DOC_PHONE_c := char(without_null_base(DOC_PHONE));
      end if;
      if is_null(DOC_OFF_HRS)
         then DOC_OFF_HRS_indic := -1;
      else DOC_OFF_HRS_indic := 0;
         DOC_OFF_HRS_c := char(without_null_base(DOC_OFF_HRS));
      end if;
      if is_null(DOC_AREA)
         then DOC_AREA_indic := -1;
      else DOC_AREA_indic := 0;
         DOC_AREA_c := char(AREA_NOT_NULL'image(without_null(DOC_AREA)));
      end if;
      if is_null(DOC_SPEC)
         then DOC_SPEC_indic := -1;
      else DOC_SPEC_indic := 0;
         DOC_SPEC_c := char(without_null_base(DOC_SPEC));
      end if;
      if is_null(DOC_BED_MAN)
         then DOC_BED_MAN_indic := -1;
      else DOC_BED_MAN_indic := 0;
         DOC_BED_MAN_c := INT(without_null_base(DOC_BED_MAN));
      end if;
      concrete_interface_enum.update_doctor (CHAR(DOC_NAME),
                        CHAR(DOC_ADDRESS), DOC_PHONE_c,
                        DOC_PHONE_indic, DOC_OFF_HRS_c,
                        DOC_OFF_HRS_indic, DOC_AREA_c, DOC_AREA_indic,
                        DOC_SPEC_c, DOC_SPEC_indic,
                        DOC_BED_MAN_c, DOC_BED_MAN_indic, SQLCODE);
      if sqlcode = DUPLICATE_VALUE_VALUE then
         RESULT := DUPLICATE_VALUE;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end UPDATE_DOCTOR;

end abstract_interface_enum;
```

Table D-14. Concrete_Interface_Enum - 1st Concrete Interface Package Specification

```
with sql_standard; use sql_standard;

package concrete_interface_enum is

   procedure open_dr_row(IN_DR_NAME : in CHAR;
                         sqlcode : out sql_standard.sqlcode_type);
   pragma interface (sql, open_dr_row);

   procedure close_dr_row(sqlcode : out sql_standard.sqlcode_type);
   pragma interface (sql, close_dr_row);

   procedure fetch_doctor(DNAME: in out CHAR; DADDRESS: in out CHAR;
            DPHONE : in out CHAR;
            DPHONE_indic : out sql_standard.indicator_type;
            DOFF_HRS : in out CHAR;
            DOFF_HRS_indic : out sql_standard.indicator_type;
```

```
          DAREA : in out CHAR;
          DAREA_indic : out sql_standard.indicator_type;
          DSPEC : in out CHAR;
          DSPEC_indic : out sql_standard.indicator_type;
          DBED_MAN : in out INT;
          DBED_MAN_indic : out sql_standard.indicator_type;
          sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, fetch_doctor);

     procedure insert_insurance(INAME: in CHAR;
                      IDED: in REAL;
                      IOV_CO: in REAL;
                      IDR_CO: in REAL;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, insert_insurance);

     procedure insert_hosp(HNAME: in CHAR;
                      HADDRESS: in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, insert_hosp);

     procedure insert_dr(DNAME: in CHAR;
                      DADDRESS: in CHAR;
                      DPHONE : in out CHAR;
                      DPHONE_indic : in sql_standard.indicator_type;
                      DOFF_HRS : in out CHAR;
                      DOFF_HRS_indic:in sql_standard.indicator_type;
                      DAREA : in out CHAR;
                      DAREA_indic : in sql_standard.indicator_type;
                      DSPEC : in out CHAR;
                      DSPEC_indic : in sql_standard.indicator_type;
                      DBED_MAN : in out INT;
                      DBED_MAN_indic:in sql_standard.indicator_type;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, insert_dr);

     procedure insert_dr_ins(DNAME: in CHAR;
                      INAME: in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, insert_dr_ins);

     procedure insert_dr_hosp(DNAME: in CHAR;
                      HNAME: in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, insert_dr_hosp);

     procedure delete_dr_in_dr(DR_NAME : in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, delete_dr_in_dr);

     procedure delete_dr_in_ins(DR_NAME : in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, delete_dr_in_ins);

     procedure delete_dr_in_hosp(DR_NAME : in CHAR;
                      sqlcode : out sql_standard.sqlcode_type);
     pragma interface (sql, delete_dr_in_hosp);
```

```
    procedure commit_transaction
                         (sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, commit_transaction);

    procedure rollback_transaction
                         (sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, rollback_transaction);

    procedure update_doctor(DOC_NAME : in CHAR;
              DOC_ADDRESS : in CHAR;
              DOC_PHONE : in CHAR;
              DOC_PHONE_indic : in sql_standard.indicator_type;
              DOC_OFF_HRS : in CHAR;
              DOC_OFF_HRS_indic : in sql_standard.indicator_type;
              DOC_AREA : in CHAR;
              DOC_AREA_indic : in sql_standard.indicator_type;
              DOC_SPEC : in CHAR;
              DOC_SPEC_indic : in sql_standard.indicator_type;
              DOC_BED_MAN : in INT;
              DOC_BED_MAN_indic : in sql_standard.indicator_type;
              sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, update_doctor);

end concrete_interface_enum;
```

Table D-15. Dbms_Specific_Enum - 1st Interface Description Text File

```
open_dr_row IN_DR_NAME in CHAR 25
open_dr_row sqlcode out sqlcode_type
close_dr_row sqlcode out sqlcode_type
fetch_doctor DNAME in_out CHAR 25
fetch_doctor DADDRESS in_out CHAR 25
fetch_doctor DPHONE in_out CHAR 11
fetch_doctor DPHONE_indic out indicator_type
fetch_doctor DOFF_HRS in_out CHAR 20
fetch_doctor DOFF_HRS_indic out indicator_type
fetch_doctor DAREA in_out CHAR  5
fetch_doctor DAREA_indic out indicator_type
fetch_doctor DSPEC in_out CHAR 11
fetch_doctor DSPEC_indic out indicator_type
fetch_doctor DBED_MAN in_out INT
fetch_doctor DBED_MAN_indic out indicator_type
fetch_doctor sqlcode out sqlcode_type
insert_insurance INAME in CHAR 20
insert_insurance IDED in REAL
insert_insurance IOV_CO in REAL
insert_insurance IDR_CO in REAL
insert_insurance sqlcode out sqlcode_type
insert_hosp HNAME in CHAR 20
insert_hosp HADDRESS in CHAR 20
insert_hosp sqlcode out sqlcode_type
insert_dr DNAME in CHAR 25
insert_dr DADDRESS in CHAR 25
insert_dr DPHONE in_out CHAR 11
insert_dr DPHONE_indic in indicator_type
insert_dr DOFF_HRS in_out CHAR 20
insert_dr DOFF_HRS_indic in indicator_type
insert_dr DAREA in_out CHAR  5
insert_dr DAREA_indic in indicator_type
insert_dr DSPEC in_out CHAR 11
insert_dr DSPEC_indic in indicator_type
insert_dr DBED_MAN in_out INT
insert_dr DBED_MAN_indic in indicator_type
insert_dr sqlcode out sqlcode_type
insert_dr_ins DNAME in CHAR 25
insert_dr_ins INAME in CHAR 20
insert_dr_ins sqlcode out sqlcode_type
insert_dr_hosp DNAME in CHAR 25
insert_dr_hosp HNAME in CHAR 20
insert_dr_hosp sqlcode out sqlcode_type
delete_dr_in_dr DR_NAME in CHAR 25
delete_dr_in_dr sqlcode out sqlcode_type
delete_dr_in_ins DR_NAME in CHAR 25
delete_dr_in_ins sqlcode out sqlcode_type
delete_dr_in_hosp DR_NAME in CHAR 25
delete_dr_in_hosp sqlcode out sqlcode_type
commit_transaction sqlcode out sqlcode_type
rollback_transaction sqlcode out sqlcode_type
update_doctor DOC_NAME in CHAR 25
update_doctor DOC_ADDRESS in CHAR 25
update_doctor DOC_PHONE in CHAR 11
update_doctor DOC_PHONE_indic in indicator_type
update_doctor DOC_OFF_HRS in CHAR 20
update_doctor DOC_OFF_HRS_indic in indicator_type
update_doctor DOC_AREA in CHAR  5
update_doctor DOC_AREA_indic in indicator_type
update_doctor DOC_SPEC in CHAR 11
update_doctor DOC_SPEC_indic in indicator_type
update_doctor DOC_BED_MAN in INT
update_doctor DOC_BED_MAN_indic in indicator_type
update_doctor sqlcode out sqlcode_type
```

Table D-16. Doctors_Intview2 - 2nd Interface Definition Procedure

```
with base_specific_domains;
use base_specific_domains;
with abstract_interface_generator;
with generator_support; use generator_support;

procedure doctors_intview2 is

   type record_names is (short_doctor_record);

   type proc_names is (increment_bedside_manner, open_dr_hosp,
                       fetch_dr_hosp, close_dr_hosp, open_spec_area,
                       fetch_spec_area, close_spec_area,
                       open_doc_ins, fetch_doc_ins, close_doc_ins,
                       open_area_ins, fetch_area_ins, close_area_ins,
                       set_transaction);

   type ok_errors is (cursor_already_open, cursor_not_open, not_found,
                       duplicate_value, fetch_not_done);

   package my_interface is new abstract_interface_generator
                               ("concrete_interfaceII",
                               "abstract_interfaceII",
                                     "dbms_specificII",
                                     (1 => doctors_def_enum_pkg,
                                     2 => insurance_def_pkg,
                                     3 => hospital_def_pkg),
                                     record_names,
                                     proc_names,
                                     ok_errors,
                                     (1001, -501, 100, -803, -508));
use my_interface;

begin
   declare
      type rec_components is (dname, daddress, dphone);
      package rec4 is new record_generator (rec_components,
                                            short_doctor_record);
         package com_41 is new rec4.component_generator
                                  (dname, dr_name_not_null);
         package com_42 is new rec4.component_generator
                                  (daddress, dr_address_not_null);
         package com_43 is new rec4.component_generator
                                  (dphone, phone_number_type);
      begin
         rec4.generate_record;
      end;

   declare
      type params is (result);
      type params_conc is (result);
      package procedure4 is new procedure_with_parameters_generator
                     (procedure_name => increment_bedside_manner,
                     parameters => params,
                     sql_statement_type => update_positioned,
                     sql_module_procedure_name =>
                                  "increment_bedside_manner",
                     params_to_concrete_procedure => params_conc,
                     valid_errors => (1 => cursor_not_open,
                     2 => fetch_not_done));
      use procedure4;
         package param1 is new params_of_error_conditions_generator
                                  (result);
      begin
         procedure4.generate_procedure;
      end;

   declare
      type params is (hos_name, result);
      type params_conc is (hos_name, result);
      package procedure10 is new
         procedure_with_parameters_generator
                     (procedure_name => open_dr_hosp,
                     parameters => params,
                     sql_statement_type => open,
                     sql_module_procedure_name => "open_dr_hosp",
                     params_to_concrete_procedure => params_conc,
                     valid_errors => (1 => cursor_already_open));
      use procedure10;
         package param1 is new params_of_domain_type_generator
                                  (hos_name, hosp_name_not_null);
```

```
        package param2 is new params_of_error_conditions_generator
                                (result);
     begin
        procedure10.generate_procedure;
     end;

  declare
     type params is (short_doctor_rec, result);
     type params_conc is (dname, daddress, dphone, result);
     package procedure11 is new
        procedure_with_parameters_generator
                (procedure_name => fetch_dr_hosp,
                parameters =>  params,
                sql_statement_type => fetch,
                sql_module_procedure_name => "fetch_dr_hosp",
                params_to_concrete_procedure => params_conc,
                valid_errors =>  (1 => cursor_not_open,
                2 => not_found));
     use procedure11;
        package param1 is new params_of_record_type_generator
                                (short_doctor_rec, shor
        t_doctor_record);
        package param2 is new params_of_error_conditions_generator
                                (result);
     begin
           procedure11.generate_procedure;
     end;

  declare
     package procedure12 is new
        procedure_without_parameters_generator
                (procedure_name => close_dr_hosp,
                sql_statement_type => close,
                sql_module_procedure_name => "close_dr_hosp");
     begin
           procedure12.generate_procedure;
     end;

  declare
     package procedure13 is new
        procedure_without_parameters_generator
                (procedure_name => set_transaction,
                sql_statement_type => close,
                sql_module_procedure_name=>"set_transaction");
     begin
           procedure13.generate_procedure;
     end;

  declare
     type params is (insurance_name, area_name, result);
     type params_conc is (insurance_name, area_name, result);
     package procedure16 is new
        procedure_with_parameters_generator
                (procedure_name => open_area_ins,
                parameters => params,
                sql_statement_type => open,
                sql_module_procedure_name => "open_area_ins",
                params_to_concrete_procedure => params_conc,
                    valid_errors =>  (1 => cursor_already_open));
         use procedure16;
            package param1 is new params_of_domain_type_generator
                        (insurance_name,ins_name_not_null);
            package param2 is new params_of_domain_type_generator
                                  (area_name, area_not_null);
            package param3 is new params_of_error_conditions_generator
                                  (result);
         begin
               procedure16.generate_procedure;
         end;

      declare
         type params is (short_doctor_rec, result);
         type params_conc is (dname, daddress, dphone, result);
         package procedure17 is new
            procedure_with_parameters_generator
                    (procedure_name => fetch_area_ins,
                    parameters => params,
                    sql_statement_type => fetch,
                    sql_module_procedure_name => "fetch_area_ins",
                    params_to_concrete_procedure => params_conc,
                    valid_errors =>  (1 => cursor_not_open,
                    2 => not_found));
         use procedure17;
            package param1 is new params_of_record_type_generator
                    (short_doctor_rec,short_doctor_record);
            package param2 is new params_of_error_conditions_generator
                                  (result);
         begin
               procedure17.generate_procedure;
         end;

      declare
         package procedure18 is new
            procedure_without_parameters_generator
                    (procedure_name => close_area_ins,
                    sql_statement_type => close,
                    sql_module_procedure_name=> "close_area_ins");
         begin
               procedure18.generate_procedure;
         end;

         declare
         type params is (insurance_name, result);
         type params_conc is (insurance_name, result);
         package procedure21 is new
            procedure_with_parameters_generator
                    (procedure_name => open_doc_ins,
                    parameters => params,
                    sql_statement_type => open,
                    sql_module_procedure_name => "open_doc_ins",
                    params_to_concrete_procedure => params_conc,
                    valid_errors => (1 => cursor_already_open));
         use procedure21;
            package param1 is new params_of_domain_type_generator
                    (insurance_name, ins_name_not_null);
            package param2 is new params_of_error_conditions_generator
```

```
                                   (result);
     begin
          procedure21.generate_procedure;
     end;

   declare
     type params is (short_doctor_rec, result);
     type params_conc is (dname, daddress, dphone, result);
     package procedure22 is new
        procedure_with_parameters_generator
                   (procedure_name => fetch_doc_ins,
                   parameters => params,
                   sql_statement_type => fetch,
                   sql_module_procedure_name => "fetch_doc_ins",
                   params_to_concrete_procedure => params_conc,
                   valid_errors =>  (1 => cursor_not_open,
                   2 => not_found));
     use procedure22;
        package param1 is new params_of_record_type_generator
                   (short_doctor_rec, short_doctor_record);
        package param2 is new params_of_error_conditions_generator
                                   (result);
     begin
        procedure22.generate_procedure;
     end;

  declare
     package procedure23 is new
        procedure_without_parameters_generator
                   (procedure_name => close_doc_ins,
                   sql_statement_type => close,
                   sql_module_procedure_name => "close_doc_ins");
     begin
          procedure23.generate_procedure;
     end;

  declare
     type params is (specialty, area_name, result);
     type params_conc is (specialty, area_name, result);
     package procedure24 is new
        procedure_with_parameters_generator
                   (procedure_name => open_spec_area,
                   parameters => params,
                   sql_statement_type => open,
                   sql_module_procedure_name => "open_spec_area",
                   params_to_concrete_procedure => params_conc,
                   valid_errors =>  (1 => cursor_already_open));
     use procedure24;
        package param1 is new params_of_domain_type_generator
                                   (specialty, specialty_not_null);
        package param2 is new params_of_domain_type_generator
                                   (area_name, area_not_null);
        package param3 is new params_of_error_conditions_generator
                                   (result);
     begin
          procedure24.generate_procedure;
     end;
```

```
      declare
         type params is (short_doctor_rec, result);
         type params_conc is (dname, daddress, dphone, result);
         package procedure25 is new
            procedure_with_parameters_generator
                        (procedure_name => fetch_spec_area,
                        parameters => params,
                        sql_statement_type => fetch,
                        sql_module_procedure_name=> "fetch_spec_area",
                        params_to_concrete_procedure => params_conc,
                        valid_errors =>  (1 => cursor_not_open,
                        2 => not_found));
         use procedure25;
            package param1 is new params_of_record_type_generator
                        (short_doctor_rec, short_doctor_record);
            package param2 is new params_of_error_conditions_generator
                                (result);
         begin
              procedure25.generate_procedure;
         end;

      declare
         package procedure26 is new
            procedure_without_parameters_generator
                        (procedure_name => close_spec_area,
                        sql_statement_type => close,
                        sql_module_procedure_name=>"close_spec_area");
         begin
              procedure26.generate_procedure;
         end;

   my_interface.generate_interface;

end doctors_intview2;
```

Table D-17. Abstract_InterfaceII - 2nd Abstract Interface Package Specification

```
with DOCTORS_DEF_ENUM_PKG;
use DOCTORS_DEF_ENUM_PKG;
with INSURANCE_DEF_PKG;
use INSURANCE_DEF_PKG;
with HOSPITAL_DEF_PKG;
use HOSPITAL_DEF_PKG;
with sql_standard;
use sql_standard;

package abstract_interfaceII is

   type valid_status_result_type is
                        (CURSOR_ALREADY_OPEN, CURSOR_NOT_OPEN,
                        NOT_FOUND, DUPLICATE_VALUE, FETCH_NOT_DONE);

   type SHORT_DOCTOR_RECORD is record
      DNAME : DR_NAME_NOT_NULL;
      DADDRESS : DR_ADDRESS_NOT_NULL;
      DPHONE : PHONE_NUMBER_TYPE;
   end record;
```

```
    procedure INCREMENT_BEDSIDE_MANNER
                         (RESULT : out valid_status_result_type);

    procedure OPEN_DR_HOSP(HOS_NAME : in HOSP_NAME_NOT_NULL;
                         RESULT : out valid_status_result_type);

    procedure FETCH_DR_HOSP
             (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                         RESULT : out valid_status_result_type);

    procedure CLOSE_DR_HOSP;

    procedure SET_TRANSACTION;

    procedure OPEN_AREA_INS
                         (INSURANCE_NAME : in INS_NAME_NOT_NULL;
                         AREA_NAME : in AREA_NOT_NULL;
                         RESULT : out valid_status_result_type);

    procedure FETCH_AREA_INS
             (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                         RESULT : out valid_status_result_type);

    procedure CLOSE_AREA_INS;

    procedure OPEN_DOC_INS
                         (INSURANCE_NAME : in INS_NAME_NOT_NULL;
                         RESULT : out valid_status_result_type);

    procedure FETCH_DOC_INS
             (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                         RESULT : out valid_status_result_type);

    procedure CLOSE_DOC_INS;

    procedure OPEN_SPEC_AREA(SPECIALTY : in SPECIALTY_NOT_NULL;
                         AREA_NAME : in AREA_NOT_NULL;
                         RESULT : out valid_status_result_type);

    procedure FETCH_SPEC_AREA
             (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                         RESULT : out valid_status_result_type);

    procedure CLOSE_SPEC_AREA;

end abstract_interfaceII;
```

Table D-18. Abstract_InterfaceII - 2nd Abstract Interface Package Body

```
with sql_communications_pkg, sql_database_error_pkg, conversions,
 concrete_interfaceII;

use sql_communications_pkg, sql_database_error_pkg, conversions;

package body abstract_interfaceII is

   use PHONE_NUMBER_OPS, OFFICE_HOURS_OPS, AREA_OPS, SPECIALTY_OPS,
                                                 BEDSIDE_MANNER_OPS;
```

```
CURSOR_ALREADY_OPEN_VALUE : constant :=  1001;
CURSOR_NOT_OPEN_VALUE : constant := -501;
NOT_FOUND_VALUE : constant :=  100;
DUPLICATE_VALUE_VALUE : constant := -803;
FETCH_NOT_DONE_VALUE : constant := -508;

procedure INCREMENT_BEDSIDE_MANNER
                     (RESULT : out valid_status_result_type) is
begin
   concrete_interfaceII.increment_bedside_manner (SQLCODE);
   if sqlcode = CURSOR_NOT_OPEN_VALUE then
      RESULT := CURSOR_NOT_OPEN;
   elsif sqlcode = FETCH_NOT_DONE_VALUE then
      RESULT := FETCH_NOT_DONE;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   end if;
end INCREMENT_BEDSIDE_MANNER;

procedure OPEN_DR_HOSP
                     (HOS_NAME : in HOSP_NAME_NOT_NULL;
                     RESULT : out valid_status_result_type) is
begin
   concrete_interfaceII.open_dr_hosp ( CHAR(HOS_NAME), SQLCODE);
   if sqlcode = CURSOR_ALREADY_OPEN_VALUE then
      RESULT := CURSOR_ALREADY_OPEN;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   end if;
end OPEN_DR_HOSP;

procedure FETCH_DR_HOSP
         (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                     RESULT : out valid_status_result_type) is
   DPHONE_c : CHAR(1..11) := (others => ' ');
   DPHONE_indic : indicator_type;
begin
   concrete_interfaceII.fetch_dr_hosp
                     (CHAR(SHORT_DOCTOR_REC.DNAME),
                     CHAR(SHORT_DOCTOR_REC.DADDRESS),
                     DPHONE_c, DPHONE_indic, SQLCODE);
   if sqlcode = CURSOR_NOT_OPEN_VALUE then
      RESULT := CURSOR_NOT_OPEN;
   elsif sqlcode = NOT_FOUND_VALUE then
      RESULT := NOT_FOUND;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   else
      assign(SHORT_DOCTOR_REC.DPHONE,
         PHONE_NUMBER_base(convert(DPHONE_c, DPHONE_indic)));
   end if;
end FETCH_DR_HOSP;

procedure CLOSE_DR_HOSP is
```

```
begin
   concrete_interfaceII.close_dr_hosp (sqlcode);
   if sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   end if;
   end CLOSE_DR_HOSP;

   procedure SET_TRANSACTION is
   begin
   concrete_interfaceII.set_transaction (sqlcode);
   if sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   end if;
end SET_TRANSACTION;

procedure OPEN_AREA_INS
                     (INSURANCE_NAME : in INS_NAME_NOT_NULL;
                     AREA_NAME : in AREA_NOT_NULL;
                     RESULT : out valid_status_result_type) is
begin
   concrete_interfaceII.open_area_ins(CHAR(INSURANCE_NAME),
         CHAR(AREA_NOT_NULL'image(AREA_NAME)), SQLCODE);
   if sqlcode = CURSOR_ALREADY_OPEN_VALUE then
      RESULT := CURSOR_ALREADY_OPEN;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   end if;
end OPEN_AREA_INS;

procedure FETCH_AREA_INS
        (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                     RESULT : out valid_status_result_type) is
   DPHONE_c : CHAR(1..11) := (others => ' ');
   DPHONE_indic : indicator_type;
begin
   concrete_interfaceII.fetch_area_ins
                     (CHAR(SHORT_DOCTOR_REC.DNAME),
                     CHAR(SHORT_DOCTOR_REC.DADDRESS),
                     DPHONE_c, DPHONE_indic, SQLCODE);
   if sqlcode = CURSOR_NOT_OPEN_VALUE then
      RESULT := CURSOR_NOT_OPEN;
   elsif sqlcode = NOT_FOUND_VALUE then
      RESULT := NOT_FOUND;
   elsif sqlcode /= 0 then
      process_database_error;
      raise sql_database_error;
   else
      assign(SHORT_DOCTOR_REC.DPHONE,
         PHONE_NUMBER_base(convert(DPHONE_c, DPHONE_indic)));
   end if;
end FETCH_AREA_INS;

procedure CLOSE_AREA_INS is
begin
   concrete_interfaceII.close_area_ins (sqlcode);
```

```
      if sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end CLOSE_AREA_INS;

   procedure OPEN_DOC_INS(INSURANCE_NAME : in INS_NAME_NOT_NULL;
                          RESULT : out valid_status_result_type) is
   begin
      concrete_interfaceII.open_doc_ins(CHAR(INSURANCE_NAME),SQLCODE);
      if sqlcode = CURSOR_ALREADY_OPEN_VALUE then
         RESULT := CURSOR_ALREADY_OPEN;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end OPEN_DOC_INS;

   procedure FETCH_DOC_INS
            (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                          RESULT : out valid_status_result_type) is
      DPHONE_c : CHAR(1..11) := (others => ' ');
      DPHONE_indic : indicator_type;
   begin
      concrete_interfaceII.fetch_doc_ins(CHAR(SHORT_DOCTOR_REC.DNAME),
                          CHAR(SHORT_DOCTOR_REC.DADDRESS),
                          DPHONE_c, DPHONE_indic, SQLCODE);
      if sqlcode = CURSOR_NOT_OPEN_VALUE then
         RESULT := CURSOR_NOT_OPEN;
      elsif sqlcode = NOT_FOUND_VALUE then
         RESULT := NOT_FOUND;
      elsif sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      else
         assign(SHORT_DOCTOR_REC.DPHONE,
            PHONE_NUMBER_base(convert(DPHONE_c, DPHONE_indic)));
      end if;
   end FETCH_DOC_INS;

   procedure CLOSE_DOC_INS is
   begin
      concrete_interfaceII.close_doc_ins (sqlcode);
      if sqlcode /= 0 then
         process_database_error;
         raise sql_database_error;
      end if;
   end CLOSE_DOC_INS;

   procedure OPEN_SPEC_AREA(SPECIALTY : in SPECIALTY_NOT_NULL;
                          AREA_NAME : in AREA_NOT_NULL;
                          RESULT : out valid_status_result_type) is
   begin
      concrete_interfaceII.open_spec_area(CHAR(SPECIALTY),
            CHAR(AREA_NOT_NULL'image(AREA_NAME)), SQLCODE);
      if sqlcode = CURSOR_ALREADY_OPEN_VALUE then
         RESULT := CURSOR_ALREADY_OPEN;
      elsif sqlcode /= 0 then
```

```
            process_database_error;
            raise sql_database_error;
         end if;
      end OPEN_SPEC_AREA;

      procedure FETCH_SPEC_AREA
               (SHORT_DOCTOR_REC : in out SHORT_DOCTOR_RECORD;
                        RESULT : out valid_status_result_type) is
         DPHONE_c : CHAR(1..11) := (others => ' ');
         DPHONE_indic : indicator_type;
      begin
         concrete_interfaceII.fetch_spec_area
                        (CHAR(SHORT_DOCTOR_REC.DNAME),
                        CHAR(SHORT_DOCTOR_REC.DADDRESS),
                        DPHONE_c, DPHONE_indic, SQLCODE);
         if sqlcode = CURSOR_NOT_OPEN_VALUE then
            RESULT := CURSOR_NOT_OPEN;
         elsif sqlcode = NOT_FOUND_VALUE then
            RESULT := NOT_FOUND;
         elsif sqlcode /= 0 then
            process_database_error;
            raise sql_database_error;
         else
            assign(SHORT_DOCTOR_REC.DPHONE
               PHONE_NUMBER_base(convert(DPHONE_c,DPHONE_indic)));
         end if;
      end FETCH_SPEC_AREA;

      procedure CLOSE_SPEC_AREA is
      begin
         concrete_interfaceII.close_spec_area (sqlcode);
         if sqlcode /= 0 then
            process_database_error;
            raise sql_database_error;
         end if;
      end CLOSE_SPEC_AREA;

   end abstract_interfaceII;
```

Table D-19. Concrete_InterfaceII - 2nd Concrete Interface Package Specification

```
   with sql_standard; use sql_standard;

   package concrete_interfaceII is

      procedure increment_bedside_manner
                        (sqlcode : out sql_standard.sqlcode_type);
      pragma interface (sql, increment_bedside_manner);

      procedure open_dr_hosp(HOS_NAME : in CHAR;
                        sqlcode : out sql_standard.sqlcode_type);
      pragma interface (sql, open_dr_hosp);

      procedure fetch_dr_hosp(DNAME: in out CHAR; DADDRESS: in out CHAR;
                        DPHONE: in out CHAR;
                        DPHONE_indic:out sql_standard.indicator_type;
                        sqlcode : out sql_standard.sqlcode_type);
      pragma interface (sql, fetch_dr_hosp);
```

```
    procedure close_dr_hosp(sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, close_dr_hosp);

    procedure set_transaction(sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, set_transaction);

    procedure open_area_ins(INSURANCE_NAME : in CHAR;
                             AREA_NAME : in CHAR;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, open_area_ins);

    procedure fetch_area_ins(DNAME: in out CHAR;
                             DADDRESS: in out CHAR;
                             DPHONE : in out CHAR;
                             DPHONE_indic: out sql_standard.indicator_type;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, fetch_area_ins);

    procedure close_area_ins(sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, close_area_ins);

    procedure open_doc_ins(INSURANCE_NAME : in CHAR;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, open_doc_ins);

    procedure fetch_doc_ins(DNAME: in out CHAR;
                             DADDRESS: in out CHAR;
                             DPHONE : in out CHAR;
                             DPHONE_indic: out sql_standard.indicator_type;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, fetch_doc_ins);

    procedure close_doc_ins(sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, close_doc_ins);

    procedure open_spec_area(SPECIALTY : in CHAR;
                             AREA_NAME : in CHAR;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, open_spec_area);

    procedure fetch_spec_area(DNAME: in out CHAR;
                             DADDRESS: in out CHAR;
                             DPHONE : in out CHAR;
                             DPHONE_indic: out sql_standard.indicator_type;
                             sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, fetch_spec_area);

    procedure close_spec_area(sqlcode : out sql_standard.sqlcode_type);
    pragma interface (sql, close_spec_area);

end concrete_interfaceII;
```

Table D-20. Dbms_SpecificII - 2nd Interface Description Text File

```
increment_bedside_manner sqlcode out sqlcode_type
open_dr_hosp HOS_NAME in CHAR 20
open_dr_hosp sqlcode out sqlcode_type
fetch_dr_hosp DNAME in_out CHAR 25
fetch_dr_hosp DADDRESS in_out CHAR 25
fetch_dr_hosp DPHONE in_out CHAR 11
fetch_dr_hosp DPHONE_indic out indicator_type
fetch_dr_hosp sqlcode out sqlcode_type
close_dr_hosp sqlcode out sqlcode_type
set_transaction sqlcode out sqlcode_type
open_area_ins INSURANCE_NAME in CHAR 20
open_area_ins AREA_NAME in CHAR  5
open_area_ins sqlcode out sqlcode_type
fetch_area_ins DNAME in_out CHAR 25
fetch_area_ins DADDRESS in_out CHAR 25
fetch_area_ins DPHONE in_out CHAR 11
fetch_area_ins DPHONE_indic out indicator_type
fetch_area_ins sqlcode out sqlcode_type
close_area_ins sqlcode out sqlcode_type
open_doc_ins INSURANCE_NAME in CHAR 20
open_doc_ins sqlcode out sqlcode_type
fetch_doc_ins DNAME in_out CHAR 25
fetch_doc_ins DADDRESS in_out CHAR 25
fetch_doc_ins DPHONE in_out CHAR 11
fetch_doc_ins DPHONE_indic out indicator_type
fetch_doc_ins sqlcode out sqlcode_type
close_doc_ins sqlcode out sqlcode_type
open_spec_area SPECIALTY in CHAR 11
open_spec_area AREA_NAME in CHAR  5
open_spec_area sqlcode out sqlcode_type
fetch_spec_area DNAME in_out CHAR 25
fetch_spec_area DADDRESS in_out CHAR 25
fetch_spec_area DPHONE in_out CHAR 11
fetch_spec_area DPHONE_indic out indicator_type
fetch_spec_area sqlcode out sqlcode_type
close_spec_area sqlcode out sqlcode_type
```

Table D-21. Doctor_Application - Ada Application

```
with DOCTORS_DEF_ENUM_PKG;
use DOCTORS_DEF_ENUM_PKG;
with INSURANCE_DEF_PKG;
use INSURANCE_DEF_PKG;
with HOSPITAL_DEF_PKG;
use HOSPITAL_DEF_PKG;
with abstract_interface_enum;
use abstract_interface_enum;
with abstract_interfaceii;
use abstract_interfaceii;
with text_io;
use text_io;
with string_pack;
use string_pack;

procedure doctor_application is
   choice : integer := 0;

   package int_io is new text_io.integer_io(integer); use int_io;
   package real_io is new float_io(float); use real_io;
   package enum_io is new enumeration_io(area_not_null); use enum_io;
```

```
    ins_rec : abstract_interface_enum.insurance_record;
    doc_rec : abstract_interface_enum.doctor_record;
    hosp_rec : abstract_interface_enum.hospital_record;
    doc_ins_rec : abstract_interface_enum.dr_ins_record;
    doc_hosp_rec : abstract_interface_enum.dr_hosp_record;
    short_doc_rec : short_doctor_record;

    doctor_name : dr_name_not_null;
    doctor_address : dr_address_not_null;
    doctor_phone : phone_number_type;
    doctor_hours : office_hours_type;
    doctor_specialty : specialty_type;
    doctor_area : area_type;
    doctor_manner : bedside_manner_type;
    the_area : area_not_null;
    bedside_manner : bedside_manner_not_null;
    hospital_name : hosp_name_not_null;
    insurance_name : ins_name_not_null;
    specialty_name : specialty_not_null;
    string_holder : string (1..100) := (others => ' ');
    last : natural;
    done : boolean := false;
    answer : string (1..1) := (others => ' ');
    trash : string (1..1) := (others => ' ');

    worked : abstract_interface_enum.valid_status_result_type;
    workedii : abstract_interfaceii.valid_status_result_type;

 begin
    set_transaction;
    text_io.put_line("WELCOME TO THE DOCTOR DATABASE");
    text_io.put_line("What operation would you like to perform?");
    text_io.put_line("Enter appropriate number");
    text_io.put_line(" ");
    text_io.put_line(" ");
    while choice /= 14 loop
       text_io.put_line("1 => Add new insurance company.");
       text_io.put_line("2 => Add new hospital.");
       text_io.put_line("3 => Add new doctor.");
       text_io.put_line("4 => Delete doctor from database.");
       text_io.put_line("5 => Commit this session.");
       text_io.put_line("6 => Rollback this session.");
       text_io.put_line("7 => Update doctor information.");
       text_io.put_line("8 => Increment bedside manner.");
       text_io.put_line("9 => View doctor information.");
       text_io.put_line("10 => View doctors from certain hospital.");
       text_io.put_line
        ("11 => View doctors honoring certain insurance.");
       text_io.put_line
        ("12 =>View doctors in an area with certain specialty.");
       text_io.put_line
        ("13=>View doctors in an area honoring certain insurance.");
       text_io.put_line("14 => End this session.");
       int_io.get(choice);
       text_io.get_line(trash, last);
       case choice is
       when 1 =>
```

```
      text_io.put("Input insurance company name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(ins_rec.iname));
      text_io.put_line("Input deductible => ");
      real_io.get(float(ins_rec.ided));
      text_io.put_line("Input office visit copayment => ");
      real_io.get(float(ins_rec.iov_co));
      text_io.put_line("Input perscription copayment => ");
      real_io.get(float(ins_rec.idr_co));
      insert_insurance(ins_rec, worked);

   when 2 =>
      text_io.put_line("Input hospital name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(hosp_rec.hname));
      text_io.put_line("Input hospital address => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(hosp_rec.haddress));
      insert_hosp(hosp_rec, worked);

   when 3 =>
      text_io.put_line("Input doctor's name(NO NULL ALLOWED) => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doc_rec.dname));
      move(strip(string_holder), string(doc_ins_rec.dname));
      move(strip(string_holder), string(doc_hosp_rec.dname));
      text_ic.put_line
       ("Input doctor's address (NO NULL ALLOWED) => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doc_rec.daddress));
      text_io.put_line
       ("Input doctor's phone number <RET> for NULL => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      if last = 0 then
         assign(doc_rec.dphone, null_sql_char);
      else
         assign(doc_rec.dphone, to_sql_char(strip(string_holder)));
      end if;
      text_io.put_line
       ("Input doctors office hours <RET> for NULL => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      if last = 0 then
         assign(doc_rec.doff_hrs, null_sql_char);
      else
        assign(doc_rec.doff_hrs,to_sql_char(strip(string_holder)));
      end if;
      text_io.put_line
       ("Input doctors area of practice <RET> for NULL => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      if last = 0 -- they want a null value then
      assign(doc_rec.darea, null_sql_enumeration);
   else
      enum_io.get(string_holder, the_area, last);
      assign(doc_rec.darea, with_null(the_area));
   end if;
   text_io.put_line
    ("Input doctor's specialty <RET> for NULL => ");
   string_holder := (others => ' ');
   text_io.get_line(string_holder, last);
   if last = 0 then
      assign(doc_rec.dspec, null_sql_char);
   else
      assign(doc_rec.dspec, to_sql_char(strip(string_holder)));
   end if;
   text_io.put_line
    ("Input doctor's bedside manner rating <RET> for NULL => ");
   string_holder := (others => ' ');
   text_io.get_line(string_holder, last);
   if last = 0 then -- they want a null bedside manner
      bedside_manner_ops.assign
                  (doc_rec.dbed_man, null_sql_int);
   else
      int_io.get(string_holder, integer(bedside_manner), last);
                  bedside_manner_ops.assign(doc_rec.dbed_man,
                  bedside_manner_ops.with_null(bedside_manner));
   end if;
   insert_dr(doc_rec, worked);
   while not done loop
      text_io.put_line
       ("Input doctor's hospital(NO NULL ALLOWED) => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doc_hosp_rec.hname));
      insert_dr_hosp(doc_hosp_rec, worked);
      text_io.put_line("More hospitals? .. answer y or n => ");
      text_io.get_line(answer, last);
      if answer = "y" then
         text_io.get_line(trash, last);
      else
         done := true;
      end if;
   end loop;
   done := false;
   text_io.get_line(trash, last);
   while not done loop
      text_io.put_line
       ("Input doctor's insurance (NO NULL ALLOWED) => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doc_ins_rec.iname));
      insert_dr_ins(doc_ins_rec, worked);
      text_io.put_line("More insurance? .. answer y or n => ");
      text_io.get_line(answer,last);
      if answer = "y" then
         text_io.get_line(trash, last);
      else
         done := true;
      end if;
      end loop;

   when 4 =>
      text_io.put_line("Input doctor's name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doctor_name));
      delete_dr_in_dr(doctor_name, worked);
      delete_dr_in_ins(doctor_name, worked);
      delete_dr_in_hosp(doctor_name, worked);

   when 5 =>
      commit_transaction;
      set_transaction;

   when 6 =>
      rollback_transaction;
      set_transaction;

   when 7 =>
      text_io.put_line("Input doctor's name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doctor_name));
      open_dr_row(doctor_name, worked);
      fetch_doctor(doc_rec, worked);
      text_io.put_line("Current address => " &
         string_pack.strip(string(doc_rec.daddress)));
      text_io.put_line("New address => <RET> if same");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      if last = 0 then
         doctor_address := doc_rec.daddress;
      else
         move(strip(string_holder), string(doctor_address));
      end if;
      if is_null(doc_rec.dphone) then
         text_io.put_line("Current phone number is NULL");
      else
         text_io.put_line("Current phone number => " &
                     strip(to_string(doc_recdphone)));
      end if;
      text_io.put_line
       ("New phone => <RET> if same, NULL for null value");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      if last = 0 then
         assign(doctor_phone, doc_rec.dphone);
      elsif strip(string_holder) = "NULL" then
         assign(doctor_phone, null_sql_char);
      else
         assign(doctor_phone, to_sql_char(strip(string_holder)));
      end if;
      if is_null(doc_rec.doff_hrs) then
         text_io.put_line("Current office hours are NULL");
      else
         text_io.put_line("Current office hours => " &
                     strip(to_string(doc_rec.doff_hrs)));
```

```
end if;
text_io.put_line
 ("New hours => <RET> if same, NULL for null value");
string_holder := (others => ' ');
text_io.get_line(string_holder, last);
if last = 0 then
   assign(doctor_hours, doc_rec.doff_hrs);
elsif strip(string_holder) = "NULL" then
   assign(doctor_hours, null_sql_char);
else
   assign(doctor_hours, to_sql_char(strip(string_holder)));
end if;
if is_null(doc_rec.darea) then
   text_io.put_line("Current area is NULL");
else
   text_io.put_line("Current area => " &
    area_not_null'image(without_null(doc_rec.darea)));
end if;
text_io.put_line
 ("New area => <RET> if same, NULL for null value");
string_holder := (others => ' ');
text_io.get_line(string_holder, last);
if last = 0 -- then they want to keep this value
   assign(doctor_area, doc_rec.darea);
elsif strip(string_holder) = "NULL" then
   assign(doctor_area, null_sql_enumeration);
else
   enum_io.get(string_holder, the_area, last);
   assign(doctor_area, with_null(the_area));
end if;
if is_null(doc_rec.dspec) then
   text_io.put_line("Current specialty is NULL");
else
   text_io.put_line("Current specialty => " &
               strip(to_string(doc_rec.dspec)));
end if;
text_io.put_line
 ("New specialty => <RET> if same, NULL for null value");
string_holder := (others => ' ');
text_io.get_line(string_holder, last);
if last = 0 then
   assign(doctor_specialty, doc_rec.dspec);
elsif strip(string_holder) = "NULL" then
   assign(doctor_specialty, null_sql_char);
else
   assign(doctor_specialty,
               to_sql_char(strip(string_holder)));
end if;
if is_null(doc_rec.dbed_man) then
   text_io.put_line("Current bedside manner is NULL");
else
   text_io.put_line("Current bedside manner rating => " &
integer'image(integer(without_null_base(doc_rec.dbed_man))));
end if;
text_io.put_line
 ("New rating => <RET> if same, NULL for null value");
string_holder := (others => ' ');
text_io.get_line(string_holder, last);
```

```
      if last = 0 then -- they want to keep this value
         bedside_manner_ops.assign
                     (doctor_manner, doc_rec.dbed_man);
      elsif strip(string_holder) = "NULL" then
         bedside_manner_ops.assign
                     (doctor_manner, null_sql_int);
      else
         int_io.get
          (strip(string_holder), integer(bedside_manner), last);
         bedside_manner_ops.assign
                     (doctor_manner,
                     bedside_manner_ops.with_null(bedside_manner));
      end if;
      update_doctor(doctor_name, doctor_address,
                     doctor_phone, doctor_hours,doctor_area,
                     doctor_specialty, doctor_manner,
                     worked);
      close_dr_row;

   when 8 =>
      text_io.put_line("Input doctor's name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doctor_name));
      open_dr_row(doctor_name, worked);
      fetch_doctor(doc_rec, worked);
      increment_bedside_manner(workedii);
      close_dr_row;

   when 9 =>
      text_io.put_line("Input doctor's name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(doctor_name));
      open_dr_row(doctor_name, worked);
      fetch_doctor(doc_rec, worked);
      text_io.put_line("Current address => " &
                     string_pack.strip(string(doc_rec.daddress)));
      if is_null(doc_rec.dphone) then
         text_io.put_line("Current phone number is NULL");
      else
         text_io.put_line("Current phone number => " &
                     strip(to_string(doc_rec.dphone)));
      end if;
      if is_null(doc_rec.doff_hrs) then
         text_io.put_line
                     ("Current office hours are NULL");
      else
         text_io.put_line("Current office hours => " &
                     strip(to_string(doc_rec.doff_hrs)));
      end if;
      if is_null(doc_rec.darea) then
         text_io.put_line("Current area is NULL");
      else
         text_io.put_line("Current area => " &
          area_not_null'image(without_null(doc_rec.darea)));
      end if;
      if is_null(doc_rec.dspec) then
```

```
            text_io.put_line("Current specialty is NULL");
         else
            text_io.put_line("Current specialty => " &
                        strip(to_string(doc_rec.dspec)));
         end if;
         if is_null(doc_rec.dbed_man) then
            text_io.put_line("Current bedside manner is NULL");
         else
            text_io.put_line("Current bedside manner rating => " &
         integer'image(integer(without_null_base(doc_rec.dbed_man))));
         end if;
         close_dr_row;

      when 10 =>
         text_io.put_line("Input hospital name => ");
         string_holder := (others => ' ');
         text_io.get_line(string_holder, last);
         move(strip(string_holder), string(hospital_name));
         open_dr_hosp(hospital_name, workedii);
         text_io.put_line ("Doctors with privileges at " &
                                    string_holder(1..last));
         text_io.put_line (" ");
         fetch_dr_hosp(short_doc_rec, workedii);
         while (workedii /= not_found) loop
            if is_null (short_doc_rec.dphone) then
               text_io.put_line
                (string_pack.strip(string(short_doc_rec.dname))
                & " " &
                string_pack.strip(string(short_doc_rec.daddress))
                & " NULL PHONE NUMBER");
               else text_io.put_line
                (string_pack.strip(string(short_doc_rec.dname))
                &  " " &
                string_pack.strip(string(short_doc_rec.daddress))
                & " " & strip(to_string(short_doc_rec.dphone)));
            end if;
            fetch_dr_hosp(short_doc_rec, workedii);
         end loop;
         close_dr_hosp;

      when 11 =>
         text_io.put_line("Input insurance name => ");
         string_holder := (others => ' ');
         text_io.get_line(string_holder, last);
         move(strip(string_holder), string(insurance_name));
         open_doc_ins(insurance_name, workedii);
         text_io.put_line ("Doctors honoring " &
                        string_holder(1..last) & " insurance");
         text_io.put_line (" ");
         fetch_doc_ins(short_doc_rec, workedii);
         while (workedii /= not_found) loop
            if is_null(short_doc_rec.dphone) then
               text_io.put_line
                (string_pack.strip(string(short_doc_rec.dname))
                & " " &
                string_pack.strip(string(short_doc_rec.daddress))
                & " NULL PHONE NUMBER");
            else
            text_io.put_line
             (string_pack.strip(string(short_doc_rec.dname))
             &  " " &
             string_pack.strip(string(short_doc_rec.daddress))
             & " " &
             strip(to_string(short_doc_rec.dphone)));
         end if;
         fetch_doc_ins(short_doc_rec, workedii);
      end loop;
      close_doc_ins;

   when 12 =>
      text_io.put_line("Input specialty => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(specialty_name));
      text_io.put_line("Input area => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      enum_io.get(string_holder, the_area, last);
      open_spec_area(specialty_name, the_area, workedii);
      fetch_spec_area(short_doc_rec, workedii);
      while (workedii /= not_found) loop
         if is_null(short_doc_rec.dphone) then
            text_io.put_line
             (string_pack.strip(string(short_doc_rec.dname))
             & " " &
             string_pack.strip(string(short_doc_rec.daddress))
             & " NULL PHONE NUMBER");
         else
            text_io.put_line
             (string_pack.strip(string(short_doc_rec.dname))
             &  " " &
             string_pack.strip(string(short_doc_rec.daddress))
             & " " &
             strip(to_string(short_doc_rec.dphone)));
         end if;
         fetch_spec_area(short_doc_rec, workedii);
      end loop;
      close_spec_area;

   when 13 =>
      text_io.put_line("Input area => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      enum_io.get(string_holder, the_area, last);
      text_io.put_line("Input insurance name => ");
      string_holder := (others => ' ');
      text_io.get_line(string_holder, last);
      move(strip(string_holder), string(insurance_name));
      open_area_ins(insurance_name, the_area, workedii);
      fetch_area_ins(short_doc_rec, workedii);
      while (workedii /= not_found) loop
         if is_null(short_doc_rec.dphone) then
            text_io.put_line
             (string_pack.strip(string(short_doc_rec.dname))
             & " " &
             string_pack.strip(string(short_doc_rec.daddress))
```

```
               & " NULL PHONE NUMBER");
           else
              text_io.put_line
               (string_pack.strip(string(short_doc_rec.dname))
               & " " &
               string_pack.strip(string(short_doc_rec.daddress))
               & " " &
               strip(to_string(short_doc_rec.dphone)));
           end if;
           fetch_area_ins(short_doc_rec, workedii);
        end loop;
        close_area_ins;

     when 14 =>null;
        when others => null;
     end case;

  end loop; -- (while choice /= 14)

end doctor_application;
```

APPENDIX E
REFERENCES

[1] M. H. Graham. 1989. *Guidelines for the Use of the SAME*. SEI Technical Report CMU/SEI-89-TR-16 and ESD-TR-89-24. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

[2] American National Standards Institute. 1986. *Database Language—SQL*. X3.135-1986.

[3] Digital Equipment Corporation. 1988. *Vax Rdb/VMS Reference Manual*. Maynard, MA: Digital Equipment Corporation.

[4] Digital Equipment Corporation. 1988. *Developing Ada Programs on Vax/VMS*. Maynard, MA: Digital Equipment Corporation.